INSTANT

FREEZE-DRIED COMPUTER PROGRAMMING IN

BASIC

2nd *Astounding!* Edition

An Active-participation & Well-tested Instructional Workbook for the Student, Absolute Beginner, & Compleat Novice.

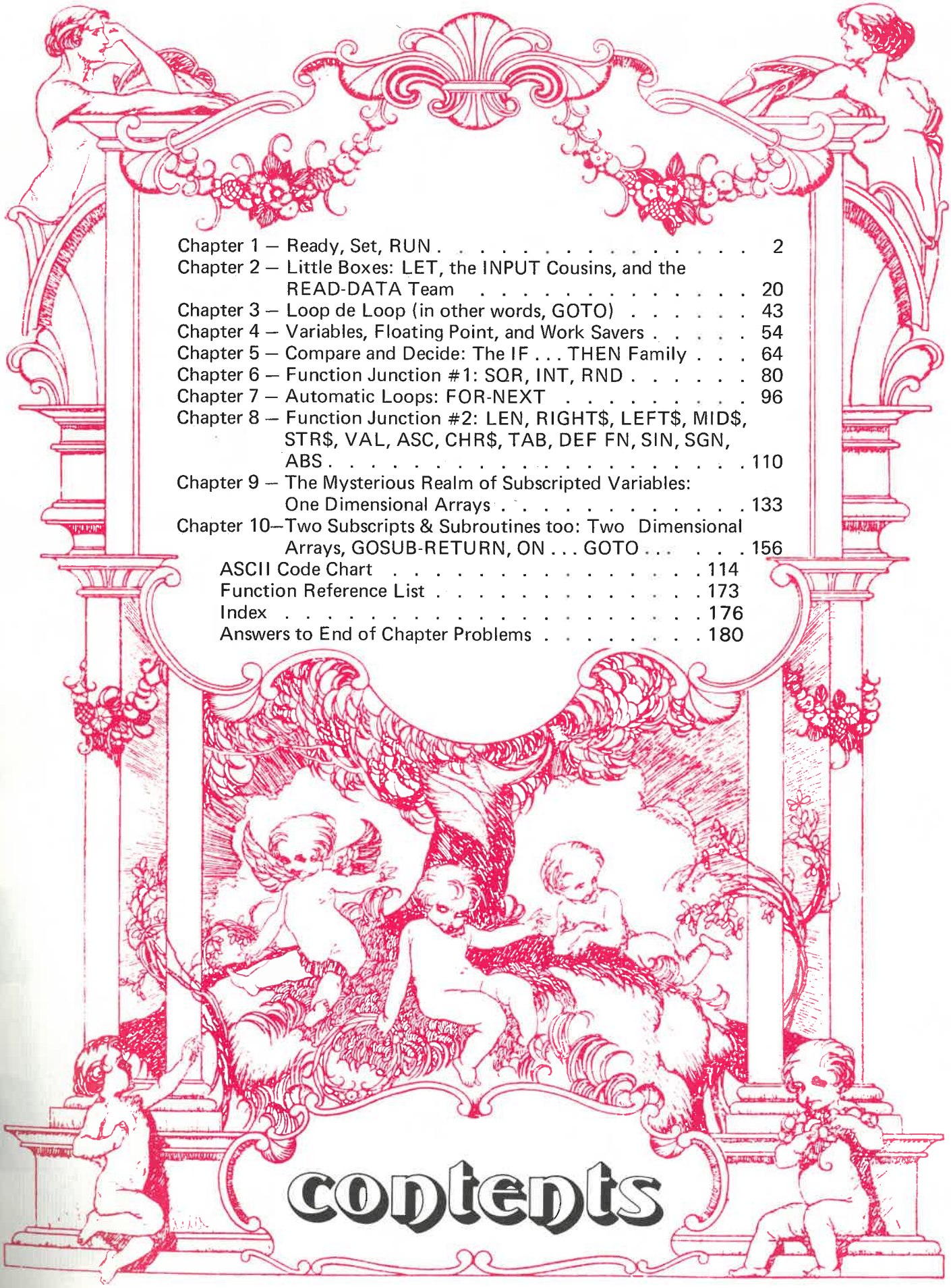
For any brand of computer using Microsoft® BASIC, including Apple®, Radio Shack®, PET® & Atari®, with annotations for North Star® BASIC and DEC BASIC Plus®.

BY
JERALD R. BROWN

If you've been wondering where to begin learning BASIC, look no further. Of the dozens of books purporting to teach computer programming, **Instant BASIC** is *the greatest* for the following reasons:

* efficiency, understandability, and economy
* microcomputer oriented for Microsoft-like versions of BASIC as used on the Apple, TRS-80, PET, Microexpander, and any brand of computer using Microsoft BASIC 80, with annotations for Northstar BASIC, Atari BASIC, DEC BASIC Plus
* no heavy math
* neat little boxed summaries of BASIC statements throughout the book
* end-of-chapter activities to see how well you're learning BASIC
* one of the smoothest and best-tested instructional sequences going!

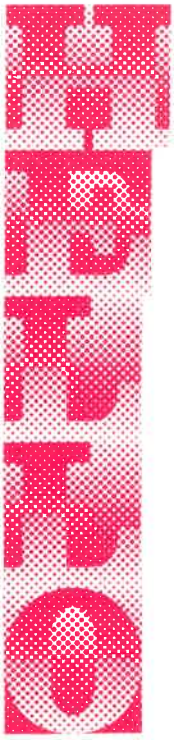
Jerry assumes that you are a first-timer, that you have had no previous programming experience at all. He keeps the information coming, but in small, discrete lumps so that you don't choke up. His examples try to keep your typing time to a minimum while maximizing learning and developing good programming techniques. That is why **Instant BASIC** is a favorite learning tool with instructors AND students from junior high to university levels. Welcome to the **2nd *Astounding!* Edition of Instant BASIC!**



contents

If you've been wondering where to begin, look no further. Of the dozens of books purporting to teach computer programming, INSTANT BASIC is *the greatest* for the following reasons:

- efficiency, understandability, and economy.
- microcomputer oriented for Microsoft-like versions of BASIC as used on the Apple, TRS-80, PET, Microexpander, and any brand of computer using Microsoft BASIC 80, with annotations for Northstar BASIC, Atari BASIC, DEC BASIC Plus, and with rumors and snide asides about other versions of good ole BASIC.
- no heavy math.
- neat little boxed summaries of BASIC statements throughout the book.
- an active-participation tutorial primer and workbook for the absolute beginner and compleat novice.
- end-of-chapter activities to see how well we've taught you BASIC.
- one of the smoothest and best-tested instructional sequences going!



We keep the information coming, but in small, discrete lumps so that you don't choke up. (Hopefully you won't choke with laughter at our attempts at humor.) We take full advantage of the interactive (immediate feedback) quality of BASIC to give you practical demonstrations and practice for fast, easy learning. You get a working familiarity with the real fundamentals of BASIC in from 6 to 26 hours, fans. Our examples try to keep your typing time to a minimum while maximizing learning and developing good programming techniques.

> Our experience teaching beginners from 6 to 76 has shown that there are three programming concepts that sometimes give first-timers problems: conditional branching (IF . . . THEN), looping or iteration (FOR . . . NEXT), and arrays (subscripted variables). We give these concepts special attention using explanations and examples we have refined and found most effective over the years.

We assume that you are a first timer, that you have had no previous programming experience at all. We assume you have ready access to a microcomputer or to a computer system running in BASIC. Those of you who have had a little experience will find the early chapters a breeze. However, for first timers, this is *not* a book to skip around in. Start here on page 1 and plow on through. This "active participation" workbook is an alternative to that headache-producing process called "digging it out of the reference manual." INSTANT BASIC is a favorite learning tool with instructors AND students from junior high to university levels!

> The first half of the book takes it really slow and easy. We encourage you (in fact, we urge and exhort to the point of browbeating...) to experiment and try out your budding programming skills beyond the examples and projects we include in the book. We want you to think of things interesting to you, and to think of how what you are learning may be applied to such things.

> Look for our Handy Reference Summaries placed throughout the book in boxes with polka dot borders like this one.

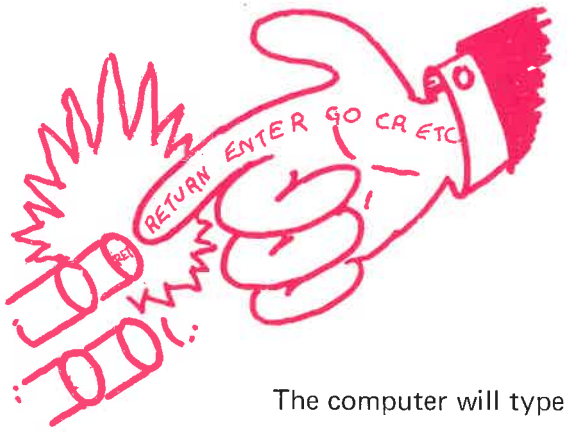
You can USE a computer (and make very good use of it, thank you!) without ever learning computer programming. Thousands of ready-made programs are available for purchase and use on various microcomputers. People also share their own programs through magazines and newsletters, computer clubs, and user's groups, which are associations of people using the same brand computer, or a specific computer language, or specific computer application. But the BASIC programmer (soon to include you) will be able to take fuller advantage of the computer, and dispell the mystery of the (supposedly) almighty, complicated, and scarey computer. Welcome to the 2nd *Astounding!* Edition of INSTANT BASIC.

This means we want you to try this on your computer, so go ahead and DO IT.

Is your machine up and running in BASIC (turned on and ready)? To get the feel of things, try typing HELLO COMPUTER

When you have finished typing HELLO COMPUTER, press the key marked CR or Carriage Return or RETURN or GO or ENTER. Despite its different names on various manufacturers' keyboards, this key is usually located along the righthand side of the keyboard. Since the keyboard I happen to be using says RETURN on this key, that's what I'll call it in this book, and you'll know to press the corresponding key on your keyboard. You'll be using it *constantly* so be sure to look for it now.

This shows a popular keyboard layout. Yours may be slightly to radically different. In either case, you'll get used to yours in short order. Some touch-sensitive "membrane" keyboards (those flat ones that have no discrete and separate keys that press down) may be quite different in layout.



The computer will type out a message in response, such as:

```
HELLO COMPUTER
?SN ERROR
```

← You type this and hit RETURN.

Computer tells you that you made an error: anything it doesn't understand is considered an error.

SYNTAX ERROR is the computer's way of saying, "Get it right, meathead!"

The point is, the computer didn't understand what you typed. People have not yet designed computers that can figure out from plain English what you want them to do. So to make the computer "do" something, we use a *computer language*, such as BASIC, to present our instructions to the computer.

Set---

...RUN it

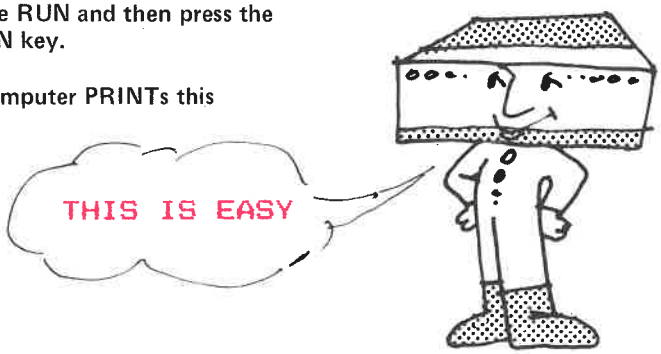
DO IT

Now type the word RUN and then press the RETURN key.

```
RUN <- * * * * * * * *
THIS IS EASY
```

You type RUN and then press the RETURN key.

The computer PRINTs this



Congratulations! You have just *entered* or typed in a *computer program* (kind of a small program, only one line of instructions), and caused the computer to RUN or *execute* your program.

Basically (if you'll excuse the pun), you told the computer to PRINT what was inside the quotation marks. Your instruction or program was this:

```
20 PRINT "THIS IS EASY"
```

The computer responded by typing or displaying

```
THIS IS EASY
```

PRINT

Print

First type NEW and hit RETURN. Then type in this program.

```
NEW

10 PRINT "WHAT"
20 PRINT "A"
30 PRINT "BREEZE"
RUN
WHAT
A
BREEZE
```

Hit RETURN at the end of each statement in the program, and also after you type RUN.

First the computer executes Line 10 and PRINTs WHAT.

Then it goes to the statement with the next higher line number, Line 20, and PRINTs A. After Line 20 the computer follows the instruction at the next higher line number, Line 30. Don't tell me, let me guess what it PRINTs.

Now you do a program like this one, using the PRINT statement to print one or more strings. Try whole sentences in each string. Don't forget the quotation marks around the string.

```
NEW
```

You don't have to *enter* or type in a program in line number order. That is, you don't have to enter line 10 first, then line 20, and then line 30. If we type in a program *out of line number order,* the computer doesn't care. It follows the line numbers, *not* the order they were *entered* or typed in. This makes it easy to *insert* more statements in a program already stored in the computer's memory. You may have noticed how we cleverly number the statements in our programs by 10's. This makes it easy to add more statements between the existing line numbers -- up to nine more statements between lines 10 and 20, for example.

READ

Let's put the PRINT statement through a few tricks so that you get some ideas of what it can do and what it can be used for. First type NEW and hit RETURN. The computer will give you the go ahead sign ... and sit there waiting.

DO IT

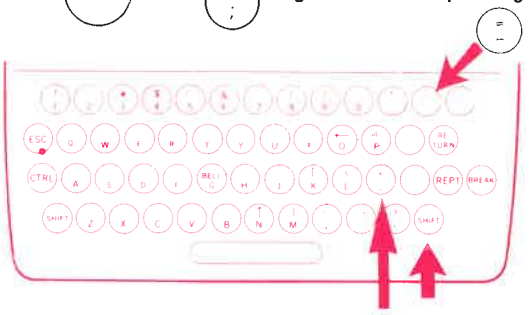
```
NEW
```

You type NEW and hit RETURN

BASIC is ready to go to work for you

Use (=/−) for the minus sign.

Use (SHIFT) and (+/;) together for the plus sign.



Then type the following:

```
10 PRINT 12+12
20 PRINT 12-12
```

You carefully type these two lines.

More Hopefully Helpful Hints –

(1) *You must hit RETURN when you finish typing an indirect statement and before you type the next line number.*

(2) *The lower case L does not substitute for the number one (1), and the letter O can't be used for a zero.*

When you RUN the program above, the computer should print the answers to the problems, like this:

```
RUN
 24
 0
```

You type RUN and hit RETURN.

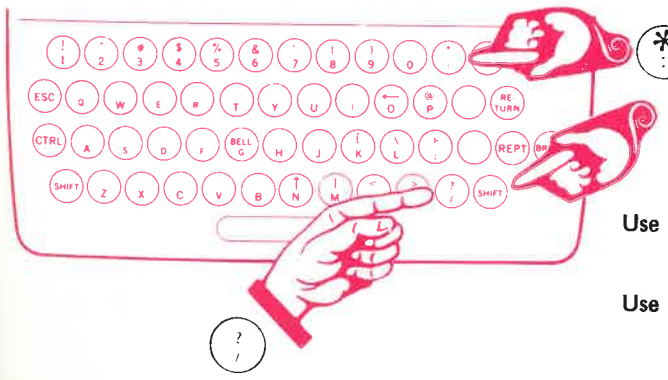
The computer does the computing and gives the results. There are 2 results, one for each line in the program.

READ

Or, as they say in computerese, the computer *evaluated* the *expressions* (12 + 12 and 12 – 12) and, per instruction, PRINTed the results.

While we're at it, here is more information on BASIC arithmetic:

- To tell the computer to add, use +
- To tell the computer to subtract, use –
- To tell the computer to multiply, use *
- To tell the computer to divide, use /



Use (SHIFT) and (*/:) together for BASIC's multiplication sign.

Use (?/) for the division sign.

Your display or printer may have 16, 24, 40, 72, 80, or even 132 *character positions* across a line. Every space across the line where a letter or number could be typed is a character position. A *character* is any number, letter, symbol, or single space. To see how many character positions across a line your printer or display can handle, type X's across a line and keep count.

For later reference, record the results of your count here.

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

Using a comma to separate items or expressions in a single PRINT statement tells the computer to space over to a *print position* at a certain place in the line before printing the next item.

You say you're a stickler for details? Very well. The comma in a PRINT statement divides the line into three or more columns, each 14 characters wide (for most versions of BASIC). After seeing a comma in a PRINT statement, the computer spaces over to the beginning of the next one of these columns in the current line before it prints the next item. By the way, many versions of BASIC count the first character position as zero rather than one. More on that after while. Meantime, you can learn to take advantage of this "automatic" spacing, which is like having permanent tab settings on a typewriter, activated by the commas separating PRINT statement items.

For the line below, read the numbers up and down, like this: 1 0 = 10 7 1 = 71

```
0123456789111111111122222222223333333333444444444455555555556666666667 7
          012345678901234567890123456789012345678901234567890123456789012345678901
```

1st Print Zone — Print Position One

2nd Print Zone — Print Position Two

3rd Print Zone — Print Position Three (last print position in the line on many video displays)

Etc. — Print Position Four — Print Position Five

The print zones or automatic tab positions vary from one system to another. Atari's BASIC, for example, has print positions at 7, 15, 23, 31, and 39, but also allows you to set your own.

PRINT

Print the values on the terminal.

(line no.) PRINT (expression) or (list)

```
20 PRINT 7 + 3
30 PRINT "INSTANT BASIC"
40 PRINT N$
```

Use ? as the short form to enter PRINT. (& in DEC BASIC Plus, ! in Northstar BASIC) Commas (,) separating items in a PRINT statement cause them to be printed or displayed across the line beginning at the next available unused print zone, each usually 14 or 15 characters wide. Semicolons (;) separating PRINT statement items cause them to be output next to each other.

```
10 PRINT C,Y,Z$
```

```
10 PRINT X,Y,Z$
20 PRINT A;B;C
```

A semicolon or comma at the end of a PRINT statement supresses the normal carriage return and line feed (cursor returns left and drops down one line).

mistrakes

READ

Do you occasionally make mistakes? We do, watch.

```
10 PTINT 2*3+4
RUN

?SN ERROR IN 10
```

We misspell PRINT.

The computer tells use we made a mistake. A check in the reference manual shows us that SN ERROR AT 10 means a syntax mistake; it didn't compute.

The point is, if we had noticed that we hit T when we meant to hit R, we could have corrected our mistake by using the back arrow key

On most keyboards designed for use with video displays, there is a key with a *left-pointing arrow* on it, and each time you press it, the cursor moves one character position to the *left*, and erases whatever was there, even a blank space.

Some BASICs check for certain kinds of errors right after you type in a line, and you don't have to RUN the program before BASIC lets you know in no uncertain terms.

DO IT

Let's clear things up.

```
NEW

10 PT←RINT 2*3+4
LIST

10 PRINT 2*3+4

RUN
 10
```

The cursor moves backwards (to the left) and cancels out the entry in one character position. Then you type the correct letter (R) and finish typing the statement with no more errors. For some printers, the back-arrow deletes the character it points to.

LIST the program.

You see? The statement is now OK.

```
NEW

10 PRINT "GET H←THE POIM ←←NT"
```

Indicates one press of the left-arrow key.

2-left-arrow keystrokes.

Deletes (takes out) the H.

Deletes the space and the M.

On video displays, the cursor moves backwards a space for each press of the "left-arrow" or backspace key. Whatever the cursor passes over while moving LEFT gets erased.

```
LIST

10 PRINT "GET THE POINT"
```

Have you ever considered taking typing lessons?

The underline key may be used just like the back-arrow or left-arrow keys on some systems, and does the same thing. Experiment! Nothing worse than a syntax error will probably befall you.

More Mistr←akes on the next pige...


```
LIST

20 PRINT "VERY"
30 PRINT "MUCH"
```

We want to replace this line.
(These two statements are still in the computer's memory.)

```
20 PRINT "TOO"
LIST
```

Type this line with the line number for the statement you want replaced.

LIST to see the modified program.

```
20 PRINT "TOO"
30 PRINT "MUCH"

RUN
TOO
MUCH
```

Our new program is far out . . . .

We Also Pull Teeth

Let's say you are typing along, ENTERING a statement and . . . suddenly . . . you notice that you have started the line with the line number of another statement, one you entered (typed in) earlier. If you just press RETURN to start over, you will replace that earlier statement with the line you are typing in now, since the second one will replace the earlier statement with the same line number. If you notice this in time (*before* you press RETURN), you can *cancel* the line you are working on, using a combination SHIFT and P key (SHIFT/P). (Exceptions are noted below.)

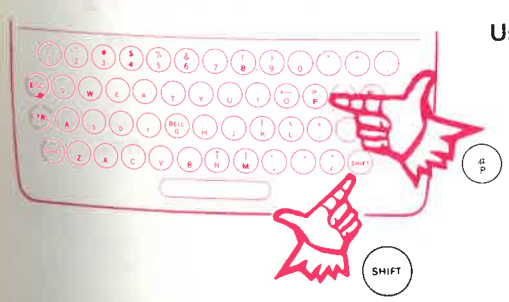
For video displays, it is very easy to just use the back-arrow key to erase all the characters in the line back to and including the mistaken or duplicated line number. On some keyboards, holding down the back-arrow key will cause the cursor to move right along (to the left, of course), erasing each character in the line that it passes over, until it gets all the way back to the left edge of the display screen. On other systems, holding the back-arrow or left-arrow key down while also pressing a REPEAT or REPT key will cause this "clean sweep" of the statement line all the way back to the beginning. Experiment!

DO IT

@

```
10 PRINT "GET THE OIUB@
```

We got frustrated and decided to take out the entire line. On a video display, the cursor drops down one line, and returns to the left side of the screen, just as if you had pressed RETURN. A printer does a *line feed* (paper goes up one line) and a *carriage return* (print head goes back to left side of the paper).



Use (SHIFT) and (@ P) together.

For Radio Shack, Northstar, others: SHIFT/P
For Atari: SHIFT/DELETE
For Apple: CONTROL/X
For DEC BASIC Plus: SHIFT/X

How BASIC Figures It.

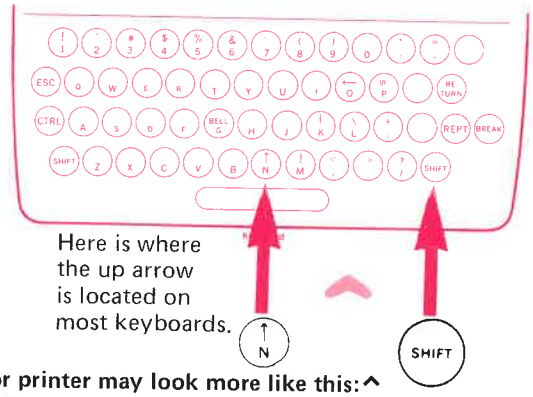
Enter and RUN the following programs. Or, if you'd rather, use *direct mode* to do each line or each arithmetic problem.

```
NEW
```

Didja or dintcha?

```
10 PRINT 2*3+4, 2*3+4*5, 2*3/4
20 PRINT 2*(3+4), (2+3)*(4+5), (2+3)/(4+5)
RUN
 10              26              1.5
 14              45              .555556

NEW

10 PRINT 2*2*2*2*2
20 PRINT 2↑5
RUN
 32
 32
```

2 x 2 x 2 x 2 x 2 is the same as 2^5 (two to the fifth power). Note the use of the up arrow ↑ in BASIC to compute the power of a number.



Here is where the up arrow is located on most keyboards.

The up arrow on your display or printer may look more like this: ^
BASIC Plus uses ** (2 asterisks) instead of up arrow.

THE RULES

1. BASIC *evaluates an expression* (does the arithmetic) by starting at the left side of the expression and working towards the right...

2. ...doing all the power (↑) computations first...†

3. ...then starting at the left again and working right, doing all the multiplications (*) and divisions (/)...

4. ...then starting once again at the left, BASIC works through the expression doing all the additions (+) and subtractions (—).

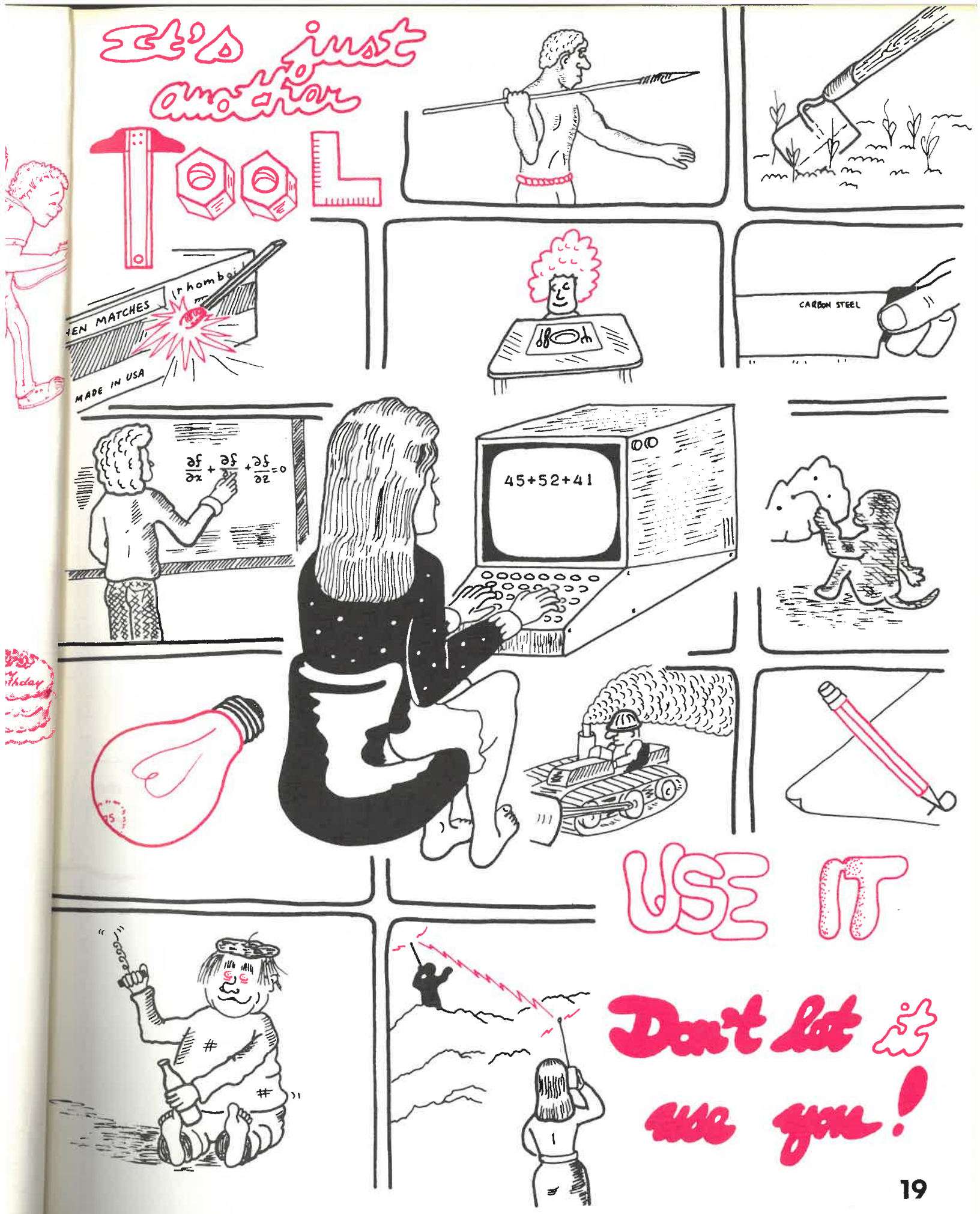
5. However, BASIC evaluates the expressions *inside* of parentheses ( ) first, following the same Rules of Precedence (who comes first) stated in 1 to 4 above.

6. If there are pairs of parentheses *within* parentheses, the evaluation or computing is done inside the inner-most set of parentheses first, then the computing is done inside the next set of parentheses, and so on and so on.

7. But don't forget: each *left* parenthesis (must have a matching *right* parenthesis), and vice versa, or BASIC will give you an error message when you try to RUN the program.

† Of course you can control the order in which power (↑) or any other calculation is done by using parentheses — see 5, 6 and 7 above. Join the Parentheses Power movement!

EXPERIMENT!


It's just another
TOOL
MADE IN USA
CARBON STEEL
45+52+41
USE IT
Don't let it use you!

Now try this one:

```
NEW

10 LET A=5
20 LET B=10
30 LET A=15
40 PRINT A-B
RUN
 5
```

What?? Two A's being assigned different values by lines 10 and 30? If you look closely, you'll see that the value assigned in Line 30 is the value (15) used in Line 40 to *evaluate* (do the arithmetic) and print the result. BASIC always uses the *last* value assigned to a variable. In effect, any new value assigned to a variable, replaces the former value in the box for the same variable. The old value is lost forever, unless a new assignment is made.

Here is a *trace* of the program as the computer goes through the program in line number order. A trace traces the path the computer uses when RUNning the program, showing the *value* of the variables after each statement is executed by the computer. The computer, in case you haven't noticed, is a fast worker, and starts to print the results of our program almost as soon as you type RUN and hit RETURN.

STATEMENTS	VARIABLES & VALUES	EXPLANATION
`10 LET A=5`	A: 5	**The value of A after the computer executes (follows the instruction) in Line 10.**
`20 LET B=10`	A: 5 B: 10	**The value of A is still 5 after the computer executes Line 20 and assigns the value 10 to variable B.**
`30 LET A=15`	A: 15 B: 10	**The old value of A is replaced by the new value 15 after the computer executes Line 30. The value of B hasn't changed.**
`40 PRINT A-B`	A: 15 B: 10	**The values of the variables don't change as the computer executes Line 40. The values in A and B are used by the computer to evaluate A – B and PRINT the result**
`RUN` ` 5`		

THIS IS A TRACE.

Your system may have a utility program that automatically traces the path your computer follows when executing a program, with line numbers as well as values of variables displayed as the RUN procedes. This is handy not only for understanding the workings of a program that works, but also for seeing where a program that doesn't work as expected is going wrong.

As you can see from that last example, a variable can only have one value at a time, and the last value assigned will be the one recorded in the little box for that variable. The previous value is replaced by the latest one assigned, and any previous value of that variable is gone forever.

DO IT

```
NEW

10 LET F=6
20 LET G=8
30 LET H=F*G
40 PRINT H
RUN
 48
```

Lots of new vocabulary and ideas in this section. Don't be afraid to READ and DO IT more than once!

Look at this *trace* to see exactly what happens when the program is RUN.

STATEMENTS	VARIABLES & VALUES	EXPLANATION
`10 LET F=6`	F: 6	Value assigned to F
`20 LET G=8`	F: 6 G: 8	Value assigned to G.
`30 LET H=F*G`	F: 6 G: 8 H: 48	Value assigned to H, using the values of F and G for the computations (doing the arithmetic).
`40 PRINT H`	F: 6 G: 8 H: 48	Computer prints the value of H. Note that all values are still in their boxes.
`RUN` `48`		All of the above happens after you type RUN and before H is printed. Fast, huh?

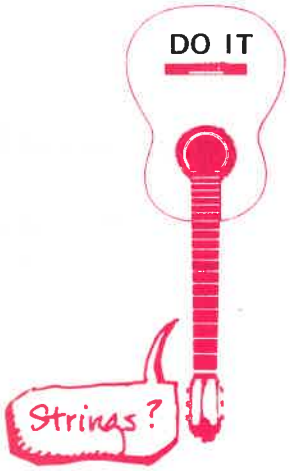
Some versions of BASIC have a TRACE instruction that shows you the line numbers of the statements as they are executed, and the order they are executed, during a RUN, even if there are no PRINT statements (such as lines 10, 20, and 30 in this program). This can be helpful in debugging a program.

THIS IS A TRACE.

"Strings in Quotes"

READ

However, there is more to variables than just putting numbers in boxes. Instead of a number, the value of a variable can be a *string*. So that BASIC knows that we are dealing with a *string variable*, the label or variable name ends with a $, for example, A$ in a LET statement. The string that is being assigned to the string variable is enclosed by quotation marks, just as in a PRINT statement.




```
NEW

10 LET C$="VERY"
20 LET E$="GOOD"
30 PRINT C$; C$; E$
RUN
VERYVERYGOOD
```

C$	VERY
E$	GOOD

Well, we could try it another way. Without typing NEW, *replace* Line 30 with a new Line 30, using commas where the semicolons were.

```
30 PRINT C$, C$, E$
RUN
VERY            VERY            GOOD
```

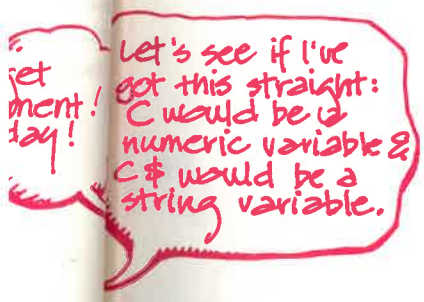
Hmmmm, still not too good. Replace Lines 10 and 30 like this

```
10 LET C$="VERY "
30 PRINT C$; C$; E$
LIST
```

see the extra space

semicolons again

```
10 LET C$="VERY "
20 LET E$="GOOD"
30 PRINT C$; C$; E$

RUN
VERY VERY GOOD
```



Did you enter (type in) this program exactly, and still got an error message when you tried to RUN it? If not, go to the next page and collect $200. For Atari and some other non-Microsoft BASICs, you must *always* inform the computer of two things at the beginning of the program:
(1) that you will be using strings assigned to string variables in the program, and
(2) the maximum number of characters that a specific string variable can be assigned. (Using *less* than the maximum number of characters you have specified is always OK.)
To do this, use the DIM (for DIMension) statement, like this one for the program above. The number in parentheses gives the maximum string length for that specific string variable's assignments.

```
5 DIM C$(5), E$(5)
```

Only if you got an error message before, add this line to the programs above and RUN again.

Note that more than one string variable can be DIMensioned in the same DIM statement, with each one separated by (what else?) a comma. See also pages 26 to 31, and 142. Since this procedure is not for most BASICs, those of you who need it will be expected to supply your own DIM statement at the beginning of all programs using string variables. (Excuse the inconvenience.) And note this: If you attempt to assign strings *longer* than the DIMensioned character count to a string variable, the "extra" characters are cut off or *truncated* to the allowable DIMensioned character count for that string variable.

```
5 DIM A$(5)
10 LET A$ = "123456789"
20 PRINT A$
RUN
12345
```

DIMensioned maximum length for A$'s string.

A 9 character string to be assigned to A$.

Oops! 6789 was left off of the string we tried to assign to A$, because the string had 4 too many characters in it. But the truncated (cut off) string "12345" *was* assigned to A$.

INPUT with "PROMPT STRING"

It's always right on time!

But that question mark isn't too informative by itself. You don't know what to respond to an INPUT question mark unless you know what the program is about. So here is how you provide a prompt or cue as to what the program needs for an INPUT.


```
NEW
10 INPUT "WHAT IS YOUR NAME"; N$
20 PRINT N$; " IS YOUR NAME."
```

Note the semicolon

Note the space

Note the semicolon

Old Atari BASIC uses the "old fashioned" method for INPUT prompts shown on page 30.

Tell the computer your name (type it in, silly, and don't forget RETURN) when it asks you.

```
RUN
WHAT IS YOUR NAME? JERALD R. BROWN
JERALD R. BROWN IS YOUR NAME.
```

It asks

You respond

N$	JERALD R. BROWN

Your INPUT string is assigned to N$.

Don't worry if the question mark was omitted by your version of BASIC (see page 30).

Notice that you must put the *prompt* string

```
"WHAT IS YOUR NAME"
```

in quotation marks, and that between the prompt string and INPUT *variable* you *must* use a semicolon (;).

```
10 INPUT "WHAT IS YOUR NAME"; N$
```

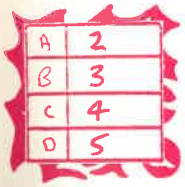
The prompt string

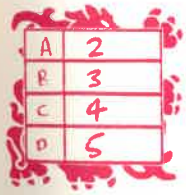
The INPUT variable.

Did N$ print less characters than you entered for your first and last names? For some versions of BASIC, you are automatically allowed to assign strings with 10 characters in them to string variables. But for longer strings (over 10 characters), you must inform the computer of the maximum length (number of characters) that each specific string variable can be assigned. To do this, go directly to page 25 and learn how to use the DIM statement, and return the $200.

more to come....

Answers for the trace on page 22.





idn't like th
Error messa
message, it
the right thi
the wrong

he
ut
a
ill

DO IT

Now LIST the program if you want to see it all together.

```
LIST

10 INPUT "WHAT IS YOUR NAME"; N$
20 PRINT N$; " IS YOUR NAME."
30 INPUT "HOW OLD ARE YOU"; A
40 PRINT N$; ", YOU ARE"; A; "YEARS OLD."
```

READ

Let us digress for just a moment here to discuss the last PRINT statement (line 40) in this program. Various versions of BASIC print numbers in slightly different ways. A leading space (where a plus sign could go) may be included for all *positive* values printed or displayed by a PRINT statement on your computer system. Try this little program:

DO IT

```
NEW                                    20 Y's.
10 ? "YYYYYYYYYYYYYYYYYYYY"
20 ? 1;2;3;4;5;6
RUN
```

Testing 1;2;3;4

READ

If your RUN looks like this:

```
YYYYYYYYYYYYYYYYYYYY
 1  2  3  4  5  6
```

then your machine prints positive values with leading AND trailing spaces.

If your RUN looks like this:

```
YYYYYYYYYYYYYYYYYYYY
 1 2 3 4 5 6
```

then your BASIC prints positive values with a leading space, but no trailing space.

If your RUN looks like this:

```
YYYYYYYYYYYYYYYYYYYY
123456
```

then guess what. No leading or trailing spaces.

Now you fix the last PRINT statement in the "name and age" program according to how your version of BASIC prints or displays positive values. If only leading spaces are included, and the RUN looked like this:

```
BILL, YOU ARE 22YEARS OLD.
```

then replace line 40 like this:

```
40 PRINT N$; ", YOU ARE"; A; " YEARS OLD."
```

Space

If neither leading nor trailing spaces are included, your RUN looked like this:

```
BILL, YOU ARE22YEARS OLD.
```

and you should use this version of line 40:

```
40 PRINT N$; ", YOU ARE "; A; " YEARS OLD."
```

Spaces

THE TRIALS AND TRIBULATIONS OF MULTIPLE INPUT VARIABLES AND NULL STRINGS


```
NEW

10  PRINT "ENTER YOUR NAME, LAST NAME FIRST,"
20  INPUT "SEPARATED BY A COMMA:";L$,F$
30  PRINT F$,L$
```

Commas separate multiple INPUT variables

1st RUN

```
RUN
ENTER YOUR NAME, LAST NAME FIRST,
SEPARATED BY A COMMA:BROWN,JERALD R.
JERALD R.          BROWN
```

PRINTed by line 30

Our two entries, one for each INPUT variable, separated by *but not including* the comma.

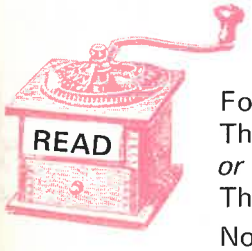
2nd RUN

```
RUN
ENTER YOUR NAME, LAST NAME FIRST,
SEPARATED BY A COMMA:
??
```

We hit RETURN without making an entry for L$.

It is very hard to see null strings!

The computer still expects an entry for the 2nd INPUT variable, F$, but we just hit RETURN again.



For the second RUN, we hit RETURN *instead of* making an entry, for both L$ and F$. The computer interprets each of these "non-entries" as instructions to assign an *empty or null string*, that is, a string with no or zero characters in it, as if L$ = "" and F$ = "". The "nothing" between each set of two quotation marks is the *null string*.

Not all versions of BASIC react as we have shown. For example, Microsoft BASIC 80 will give you the error message

? REDO FROM START

if you attempt to enter for an INPUT statement:

(1) too few values or strings.
(2) too many values or strings.
(3) the wrong type of data, that is, a string where the INPUT statement expected a numeric value for a numeric variable.

In other versions of BASIC, hitting RETURN without making an entry can (mistakenly!) assign a zero if the variable was a numeric one, or a *null string* for a string variable. Later you can learn how to "trap" these and other kinds of INPUT entry errors.

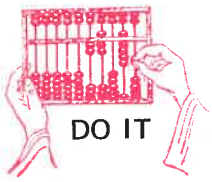
THE INVISIBLE NULL STRING


```
NEW

10  INPUT "HOW OLD ARE YOU?";A$
20  PRINT "YOU ARE";A$;"YEARS OLD."

RUN
HOW OLD ARE YOU?
YOU AREYEARS OLD.
```

No spaces inside the quotes.

We hit RETURN without making an entry.

The null string assigned to A$ doesn't take up much space, does it?



It's a Mathematical Whiz!

1st RUN ▷

```
RUN
HOW OLD ARE YOU?72
YOU ARE 72 YEARS OLD.
WHAT YEAR IS THIS?1983
IF THIS IS 1983 THEN YOU WERE BORN ABOUT
1911
```

A	72
Y	1983

OK here. ➔

2nd RUN ▷

```
RUN
HOW OLD ARE YOU?TEN
?REENTER
HOW OLD ARE YOU?10
YOU ARE 10 YEARS OLD.
WHAT YEAR IS THIS?1983
IF THIS IS 1983 THEN YOU WERE BORN ABOUT
1973
```

This BASIC will not assign the string TEN to numeric variable A, but it does give you another chance.

A	10
Y	1983

DOUBLE or ??

We don't recommend using more than one INPUT variable per INPUT statement, but there are always exceptions. In the following program, both height (inches) and weight (pounds) are entered in response to one INPUT statement for assignment to two variables.


```
NEW

10  PRINT "ENTER YOUR HEIGHT (INCHES) AND WEIGHT (LBS)"
20  INPUT "SEPARATED BY A COMMA:";H,W
30  PRINT "HEIGHT: ";H * 2.54;" CENTIMETERS"
40  PRINT "WEIGHT: ";W * .4536;" KILOGRAMS"
```

1st RUN ➔

```
RUN
ENTER YOUR HEIGHT (INCHES) AND WEIGHT (LBS)
SEPARATED BY A COMMA:69,145
HEIGHT: 175.26 CENTIMETERS
WEIGHT: 65.772 KILOGRAMS
```

We followed directions.

2nd RUN ➔

```
RUN
ENTER YOUR HEIGHT (INCHES) AND WEIGHT (LBS)
SEPARATED BY A COMMA:69
??145
HEIGHT: 175.26 CENTIMETERS
WEIGHT: 65.772 KILOGRAMS
```

We didn't follow directions, but the patient computer displays double question marks and waits for the 2nd entry.

3rd RUN ➔

```
RUN
ENTER YOUR HEIGHT (INCHES) AND WEIGHT (LBS)
SEPARATED BY A COMMA:
?REENTER
SEPARATED BY A COMMA:
?REENTER
SEPARATED BY A COMMA:
```

We stubbornly refused to make an entry, and just kept hitting RETURN.

Our BASIC refuses to continue without our INPUT entry, being equally stubborn . . .

INPUT Tests

INPUT TEST PAGE

	OK	ERROR	COMMENT
COM,MA			
SEMI;COLON			
CO:LON			
PER.IOD			
EX!CLAIM!			
A'POS'TRO'PHY			
AMP&ER&SAND			
NUM#BER			
SLA/SH			
AST*R*SK			
QU"OT"ES			
"QUOTES"			
"QU"OT"ES"			
"COM,MA"			
"SEMI;COLON"			
"CO:LON"			
"PER.IOD"			
"EX!CLAIM!"			
"A'POS'TRO'PHY"			
"AMP&ER&SAND"			
"NUM#BER"			
"SLA/SH"			
"AST*R*SK"			
"NOW," HE SAID.			
SHE ASKED, "WHY?"			

Your own tests go here.

Have you experimented today?

Try out all of these test strings, which include punctuation marks as part of a string of letters. RUN the test program for each string. Take pen in hand and use the chart to keep a record of the results, for your own future reference. Note that the second half of the list is the same as the first half, but with quotation marks enclosing the outside of the test strings. If your BASIC doesn't have LINE INPUT, then substitute INPUT or LINPUT in line 10.

DIM if needed.

```
NEW

10 LINE INPUT "ENTER TEST STRING: "; T$
20 PRINT T$
RUN
ENTER TEST STRING:
```

hework together to assign values and strings to variables.

As you can see from these examples, the READ statement will assign the value or string in the DATA statement to its variable (the READ variable). No matter where the DATA statement is placed in the program (first, last, or in the middle), the computer assigns the first item in the first DATA statement to the first READ variable.


```
NEW

10 READ A$, B$, C$
20 PRINT C$, B$, A$
30 DATA GREAT, THIS, AIN'T
RUN
AIN'T          THIS          GREAT
```

READ

Looking at the last example, notice that the items in a DATA statement are separated by *commas*, but there is *no comma* after the last item.

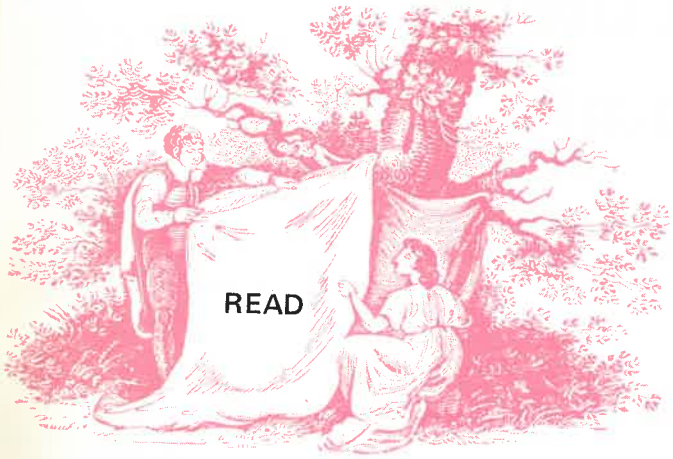
> "Users of Northstar-style BASIC will have to enclose *all* strings in DATA statements in quotation marks, as well as separate the items with commas."

Note commas

Note lack of comma

```
30 DATA GREAT, THIS, AIN'T
```

Note that spaces are not counted as part of the string unless the space is between words in the same string item. *Leading* spaces are ignored. *Trailing* spaces may be ignored, or included as an "invisible" part of the string item, depending on your version of BASIC. But *imbedded* spaces, such as those between words in one string item, are part of the string.



Like we said, the READ statement won't pick and choose among the items in a DATA statement. The items in the DATA must be in the same order as the variables — numbers only for regular variables and strings for string variables (the ones with the $ after the letter of the alphabet).

Another thing you should know about DATA statements: Even if several READ statements are used in a program they take their values in turn from the same DATA statement. When all the DATA in one DATA statement is used up, that is, when all the items have been assigned just once to variables, the computer goes on to the next DATA statement. Even if they are located in different places in a program, the computer looks for DATA statements in line number order, and takes the DATA statement with the smallest line number as the first one.

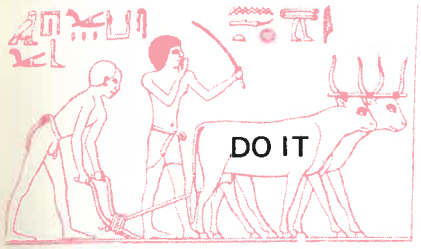
How's your typing?


```
NEW

                                   Don't forget ? for PRINT.
10 READ A
20 PRINT A
30 READ B
40 PRINT B
50 READ C
60 PRINT C
70 READ D
80 PRINT D
90 READ A
100 PRINT A
900 DATA 832, 5009, 32
910 DATA 581, 200
RUN
 832    <-- Value of A, from 900 DATA
 5009   <-- Value of B, from 900 DATA
 32     <-- Value of C, from 900 DATA
 581    <-- Value of D from 910 DATA
 200    <-- New value of A from 910 DATA
```

SPACED OUT STRINGS

Another time you must use quotation marks in DATA statements is when you want one or more spaces *before* or after the characters in the string. As we saw earlier, spaces between words or characters are included in the string.

It's just if you want spaces *before* or *after* the string that you must use quotation marks in a DATA statement. Check it out.



Type in this program, then RUN it and its several modifications.

```
NEW

10 READ A$, B$, C$
20 PRINT C$; B$; A$
30 DATA GREAT,THIS,AIN'T
RUN
AIN'TTHISGREAT
```

← three strings in a DATA statement tra la . . .

← not so great really . . .

```
30 DATA GREAT, THIS, AIN'T
RUN
AIN'TTHISGREAT
```

← replace Line 30 with a DATA statement that has spaces before or after the string items.

← the spaces were not included in the strings assigned to the string variables. But do not despair...

```
30 DATA GREAT, " THIS ", AIN'T
RUN
AIN'T THIS GREAT
```

{ replace Line 30 again with an item of DATA that uses quotation marks around the string item to tell the computer to include spaces before and after the word as part of the string.

It worked!

```
LIST

10 READ A$, B$, C$
20 PRINT C$; B$; A$
30 DATA GREAT, " THIS ", AIN'T
```

The computer is informed that we want everything inside the quotation marks included in this string, and that means spaces too.

loopdeloop

Or in other words, GOTO...

Enter this little gem in your trusty computer: (or was that your testy computer . . .)

```
NEW
```

NEW ← Clear the computer's mind to concentrate on your next instructions.

```
10 PRINT "THIS IS A LOOP."
20 GOTO 10
```

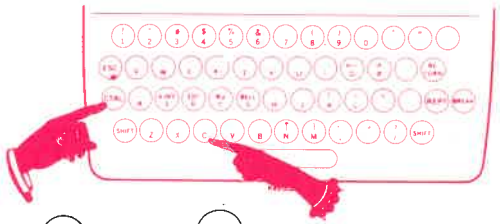
GOTO 10 tells the computer to "go to" the statement with line number 10, and to continue RUNning the program from there in normal line number order.

In almost all BASICs you can (in some you have to) type GOTO as one word, with no space between GO and TO.

Wait! Stop! Halt! Cease! Desist!

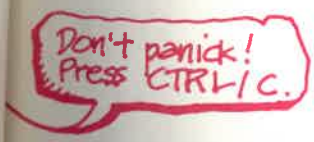
Before you RUN it, look for the CONTROL (CTRL) key, and the key of C.

Found them? OK, now RUN the program. When you get tired of watching the output, depress and hold down the CONTROL key, and while holding it down, press the C key *at the same time*, then release both keys. This will stop the computer from continuing to forever RUN this unending program.



Press (CTRL) and (C) together!


```
RUN
THIS IS A LOOP.
THIS IS A LOOP.
THIS IS A LOOP.
THIS IS A LOOP.
THIS IS A LOOP.
THIS IS A LOOP.
THIS IS A LOOP.
THIS IS A LOOP.

BREAK IN 10
```



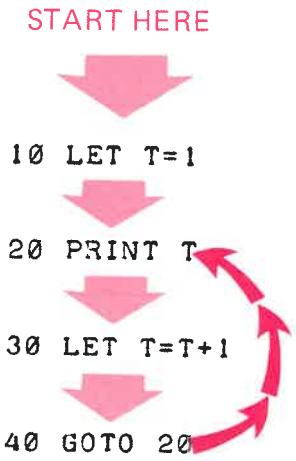
We pressed CTRL and C together.

The computer tells you at what line number CONTROL/C broke into the program. You have to be real quick on the CONTROL/C to stop a RUN of this program before it has filled the display screen from top to bottom and beyond. Try RUNning the program several times and see if you can stop the RUN before the computer has done 20 loops.

The computer executed Line 10 over and over again, because the GO TO statement in Line 20 told the computer to go back to Line 10 every time it finished executing Line 10 and got to Line 20 again. This is an *infinite loop* — it just goes on running in circles, repeating itself, forever, or until some clever person presses CONTROL and C at the same time.

Use CONTROL/C any time you wish to interrupt or stop the computer from RUNning a program. On some computers, the BREAK key or the RESET key may perform this task, but be careful with these. They may not only *stop* the program, but also *erase* it or even erase BASIC, so that it has to be reloaded or restarted.

Let's follow along as the computer RUNs that last program. Follow the arrows through the program.



GOTO ...

This is a loop. It is a "forever" loop.
It goes on and on and on and on and on and on and on and on and on ...
when the program is RUN
until you stop it with CONTROL/C.

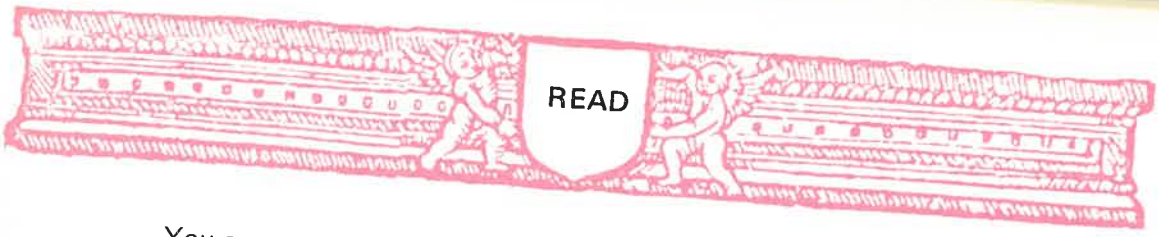
Here's another way to look at it, a *trace* of the program. Remember, a *trace* traces the path the computer takes through the program it is working on, and shows you what values are assigned to the variables at any step in the program. In the column marked T we show the value of T after the statement on the same line has been carried out by the computer. In other words, we show what value is in the box labelled T after each statement has been executed. You can think of it as always being the same little box for the same variable T — only the value assigned to T changes from time to time.

STATEMENT	VARIABLES & VALUES	EXPLANATION
10 LET T=1	T 1	Assign T the value 1
20 PRINT T	T 1	Print the value of T.
30 LET T=T+1	T 2	Increase T by 1 (add 1 to the old value of T).
40 GOTO 20	T 2	Go to beginning of loop.
20 PRINT T	T 2	Print the value of T.
30 LET T=T+1	T 3	Increase T by 1.
40 GOTO 20	T 3	Go to beginning of loop.
20 PRINT T	T 3	Print the value of T.
30 LET T=T+1	T 4	Increase T by 1.
40 GOTO 20	T 4	Go to beginning of loop.

etcetera, etcetera, etcetera

Some versions of BASIC have a TRACE instruction that shows you the line numbers of the statements as they are executed, and the order they are executed, during a RUN, even if there are no PRINT statements. This can be helpful in debugging a program.

THIS IS A TRACE



You can see that after the computer got past 65536, it started printing the results like this: 4.29497E+09. This is called *floating point notation,* and it is just a shorthand method of expressing very large numbers or very small decimal fractions. Floating point notation has two parts to it:

The E shows where the exponent starts. The exponent can be positive or negative; you'll always find a + or – after the E to clue you.

First, some examples of numbers written in the usual way, and then in floating point notation.

One billion

ordinary notation: 1000000000 or if you prefer 1,000,000,000

floating point: 1E+09

exponent

mantissa

Volume of the earth in bushels

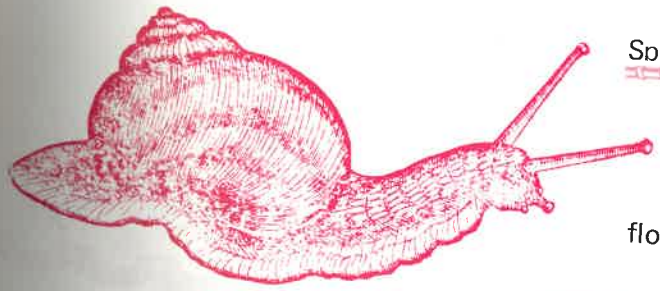
In ordinary notation we could use commas: 31,708,000,000,000,000,000,000

or 31708000000000000000000

Floating point notation: 3.1708E+22

exponent

mantissa



Speed of a snail in miles per second

ordinary notation: .0000079

floating point notation: 7.9E–06

exponent

mantissa

The version of BASIC you are using may round off or *truncate* a number after 6,7,8, or more significant digits, adding zeroes as necessary to maintain the decimal point location. We needn't get into technicalities at this point, but that is why your computer may display numbers not quite matching those in the RUN shown below. If your curiosity is killing you, the BASIC reference manual for your system may give you some clues as to what to expect in the way of number accuracy, and even the choices you may have. An Integer only version of BASIC may not be able to produce decimal fractions at all.

```
NEW

10 PRINT 1000000000000, 123456789, 3.987654321
20 PRINT .000009, .0000000654321, .700007007
30 PRINT 345.3456E4, .3E-9, 6.666666E-4
RUN
 1E+12          1.23457E+08    3.98765
 9E-06          6.54321E-08    .700007
 3.45346E+06    3E-10          6.66667E-04

LIST

10 PRINT 1000000000000, 123456789, 3.987654321
20 PRINT .000009, .0000000654321, .700007007
30 PRINT 345.3456E4, .3E-9, 6.666666E-4
```

Your BASIC may change the numbers to floating point when the program is listed, or they may stay in the same form as you entered them.

So in the future you should understand what happened if the computer gives you a result in E notation.

+ + + + + + + + + * * * * * * * * * ÷ ÷ ÷ ÷ ÷ ÷ ÷ ÷ ÷ ÷ ÷ ÷

Because of the way the computer does arithmetic (not like you and me, for sure), very small inaccuracies sometimes result from its calculations, which may show up as, say, 49.000001 or 48.999999, where you would have expected a computed result of 49 even. It's not your programming, it's the machine. And here I bet you thought the computer was as accurate as it is fast!

≈ ≈

Have you experimented today?

GOTO

Causes the program to jump or branch to specified statement.

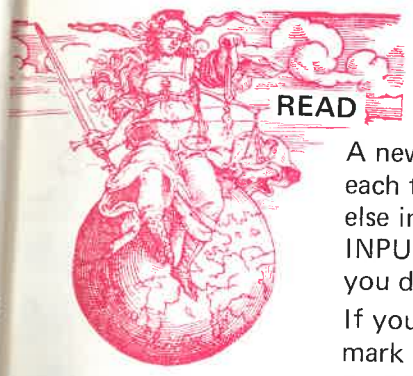
(line no.) GOTO (line number to execute next)

```
50  GOTO 10
```

DO IT HERE!

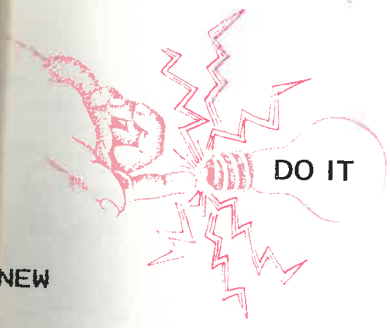
DO IT NOW!

Using INPUT to Control Displays of an Infinite GOTO Loop.



A new null string (although it's tough to see any difference) gets assigned to R$ each time you hit RETURN. Since the INPUT variable R$ isn't used anywhere else in the program, it doesn't matter what gets assigned to R$. The point is that INPUT stops the further execution of program, in mid-loop so to speak, until you do something at the keyboard.

If your version of BASIC suppresses (doesn't print) the automatic INPUT question mark when you include a prompt string, then you can use a *null string* for the *prompt string*, and nothing will be printed or displayed when the INPUT statement is executed. But the job of controlling the output display speed is still accomplished. Those with LINE INPUT available can use that statement instead.



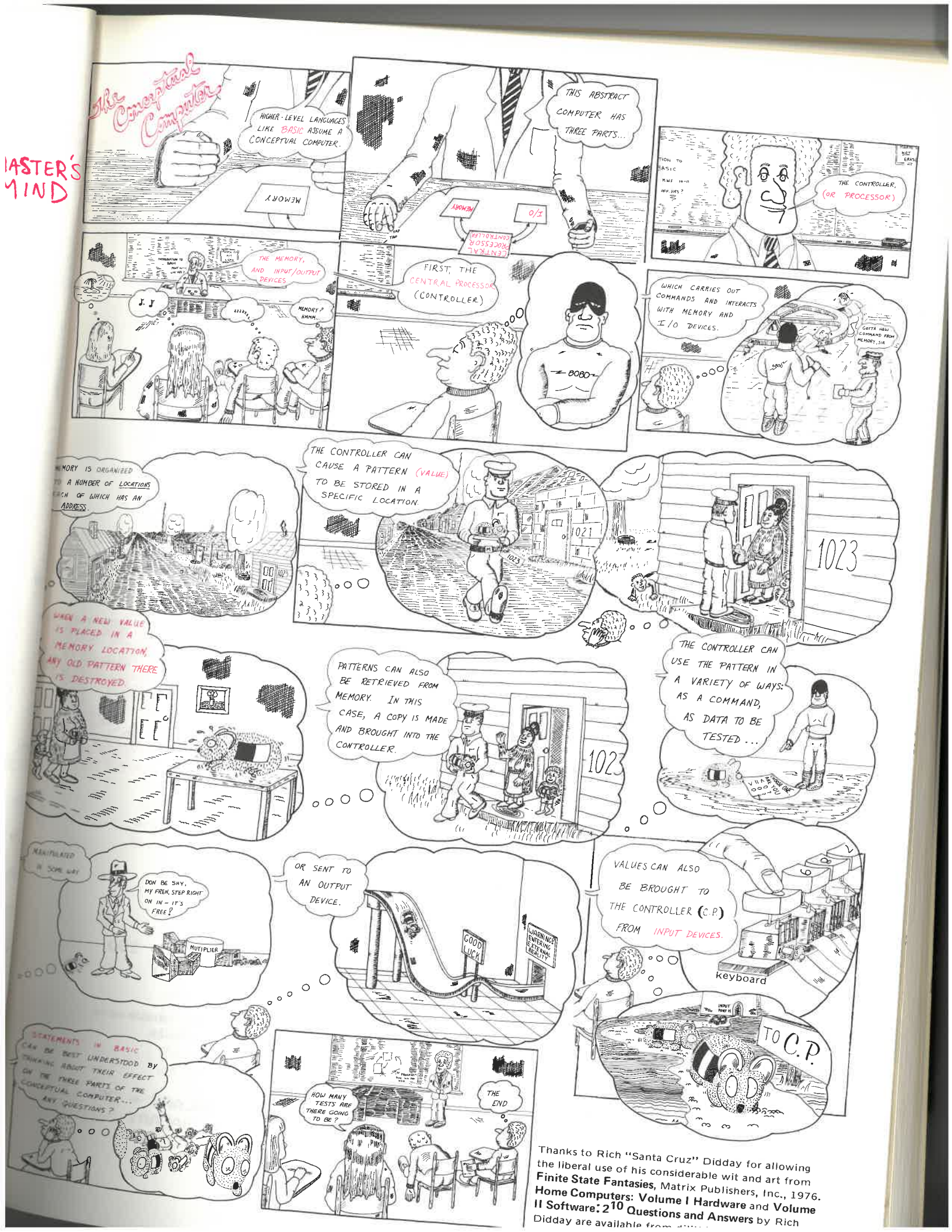

```
INPUT "";R$
```

Even though the prompt string is an invisible null string, it still suppresses the INPUT question mark for most BASICs.

```
NEW

10  LET T =  - 30
20  PRINT "PRESS 'RETURN' FOR NEXT DISPLAY"
30  INPUT "";R$
40  PRINT T;" DEGREES FAHRENHEIT = ";5 / 9 * (T - 32);" DEGREES CELSIUS"
50  LET T = T + 1
60  GOTO 30

RUN
PRESS 'RETURN' FOR NEXT DISPLAY

-30 DEGREES FAHRENHEIT = -34.4444445 DEGREES CELSIUS

-29 DEGREES FAHRENHEIT = -33.8888889 DEGREES CELSIUS

-28 DEGREES FAHRENHEIT = -33.3333333 DEGREES CELSIUS

-27 DEGREES FAHRENHEIT = -32.7777778 DEGREES CELSIUS

-26 DEGREES FAHRENHEIT = -32.2222222 DEGREES CELSIUS

BREAK IN 30
```

A user friendly program (one that is helpful to the person at the controls or keyboard) would include these statements as well:

```
22  PRINT "TO QUIT, PRESS 'CONTROL' AND 'C' TOGETHER"
23  PRINT "AND THEN PRESS 'RETURN'"
```



Thanks to Rich "Santa Cruz" Didday for allowing the liberal use of his considerable wit and art from **Finite State Fantasies**, Matrix Publishers, Inc., 1976. **Home Computers: Volume I Hardware** and **Volume II Software: 2^{10} Questions and Answers** by Rich Didday are available from

READ

However, there is a definite catch in using more than two characters in a variable name: many versions of BASIC *only* look at the first *two* characters in a long variable name. This can lead to interesting results.

DO IT

Try this program using names for variables.

```
NEW

10 LET JOHN=15
20 LET FRUMP=13
30 LET GERTRUDE=14
40 PRINT JOHN; FRUMP; GERTRUDE
50 PRINT JOE; FRANNY; GENE
RUN
 15   13   14
 15   13   14
```

JO	15
FR	13
GE	14

Line 40 printed this.

But so did Line 50!

"Yet another good idea, though unfortunately not fully implemented, is the use of datanames. In most high level languages, and assemblers for that matter, one can call a spade a spade. If you are using a variable to store a total you can call it TOTAL. This helps the programmer to remember what goes where. In this BASIC one can call a variable FRAN, if it helps, but you must proceed with considerable caution thereafter. Only the first two letters are checked so any later dataname using the same two will be confused. And you had best not use TOTAL at all. It contains the reserved word TO and any embedded reserved word will cause a syntax error. They are not always easy to spot and MITS BASIC only checks for this at RUN time, a serious weakness in any BASIC and you could find that you have to change every occurrence of a bad dataname. So, if you are starting on a long program, try the proposed dataname first!"

Reprinted from the article **Altair BASIC** *by Keith Britton and Bob Mullen,* **Peoples Computer Company**, *Vol. 4 No. 2 (Sept. 1975).*

&Prevent Unsightly Callouses!

But leaving out LET isn't the only shortcut our spiffy new BASIC provides the programmer with callouses on his/her typing fingers. Here's the short form for entering a PRINT statement:?


```
NEW

10 PRINT "HELLO GOOD LOOKING"
20 ? "USING ? FOR PRINT"
RUN
HELLO GOOD LOOKING
USING ? FOR PRINT
```

For DEC BASIC Plus, use the "and" or ampersand sign (&) instead of "?". For Atari BASIC, use the exclamation point (!) as the shorthand for PRINT.

? means PRINT when entering

Next, LIST the program and note what replaces the question mark in Line 20 of the LISTing.

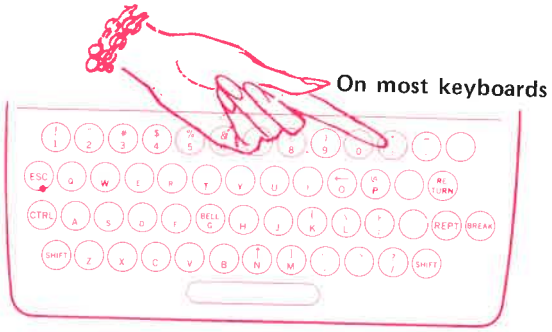
```
LIST

10 PRINT "HELLO GOOD LOOKING"
20 PRINT "USING ? FOR PRINT"
```

You use ? when you enter the program (saves typing). But when you LIST the program, the smart computer prints PRINT. (Not true for some dumb computers.)

Our next little program shows how you can pack and crunch the instructions you are giving to the computer into fewer statements, by using *multiple statements* in a line.

Use (:) (colon) to separate statements.

On this style keyboard, *don't* use SHIFT with this key, or you'll get the asterisk (*) instead of the colon (:), and an instant program bug.



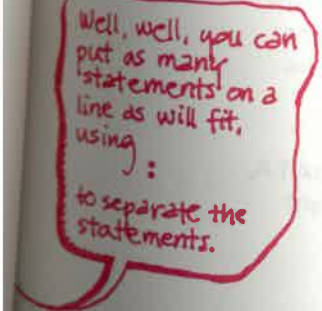

```
NEW

10 A=5 : B=10 : C=15 : D=20
20 ? A+B+C+D
RUN
 50
```

4 LETless LET statements in one line!

The ? form for PRINT, remember?

```
LIST

10 A=5 : B=10 : C=15 : D=20
20 PRINT A+B+C+D
```

Before your very eyes, a ? becomes PRINT!



Make A REMark

Another statement which must stand alone, or else be the *last* statement in a multiple statement line, is the REMark statement.


```
NEW

10 REM-SET X EQUAL TO 5 : X=5
20 ? X
RUN
 0
```

Computer never sees X = 5, because when it comes to a REMARK, it skips on to the next line in the program automatically.

Since no other value was assigned to X, the computer assumes that X = 0, and therefore, prints 0. Did you know that? An undefined variable equals zero.

```
NEW

10 X=5 : REM-SET X EQUAL TO 5
20 ? X
RUN
 5
```

This time the computer executes X = 5 *before* it sees REM. Then it skips on to the next line in the program and PRINTs the value assigned to X.

REMark statements are notes to the person reading a LISTing (or writing!) a program. They are used to explain what that line or section of a program does. REM (for remark) statements are often used to tell where a *subroutine* starts. A subroutine is a section of a program of one or more lines that performs a part of the total job of a program. You'll hear more of subroutines as we get into longer and more complex programs. Meanwhile, if you want to *document* (help explain) your program or tell the reader what your program does, just make a REMark.

Try multiple statements per line in *direct* or *immediate mode*, too!

SINS OF OMISSION
and other errors in my ways-

A word from your friendly author:

All the spaces I leave in BASIC statements are not needed: NO space is needed after the line number, nor after the BASIC statements such as PRINT, READ, DATA, INPUT or other instructions, nor after the items in a PRINT statement (after commas or semicolons). Spaces are NOT needed between DATA statement items either. Each space uses up a little of your computer's memory, but this is not a problem unless your programs are very long or have a lot of DATA values or a lot of strings. I put in those spaces just to make it easier for you to see what is happening in each statement. And remember, if you wish to save on typing, you can leave out REMark statements.

In most versions of BASIC, you do not need to initialize variables. Initialize means assigning all variables their first value. If you don't initialize in your program (using LET, INPUT, or READ), then most BASICs automatically assign a value of zero to a variable the first time it is encountered when the program is RUN. This is an invisible process—that is, it doesn't show in the program itself. If you are sharing your programs with other people, it is considered good programming practice to initialize the variables in the program, rather than letting the computer do it for you. This can avoid confusion to somebody else using your program, or even to you if you look back at your program a while after you first write it.

Refer back to previous explanations and rules of BASIC, for help in writing programs and help in debugging your program if it doesn't RUN without error messages.

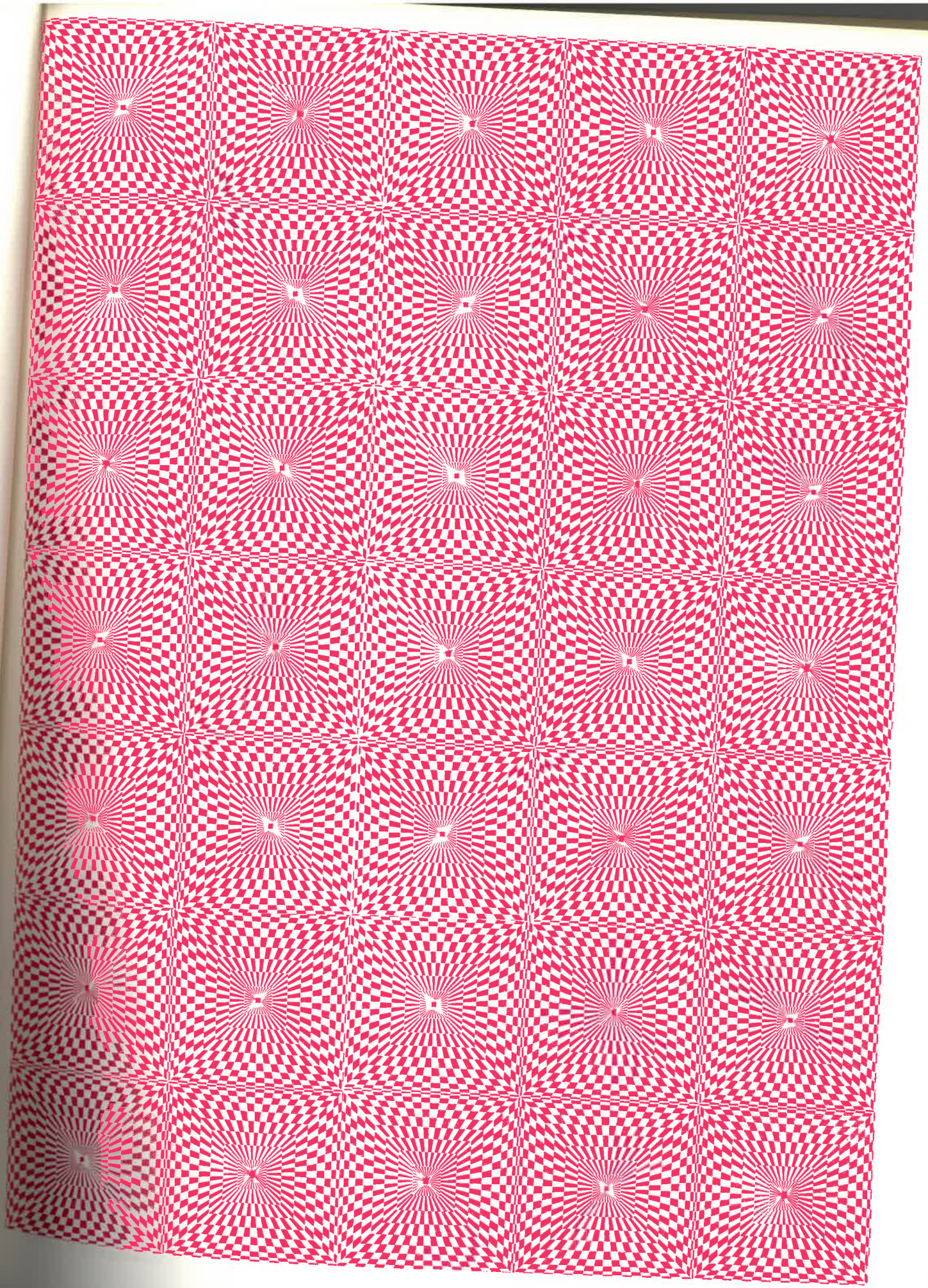
You can tell which programs I couldn't type in correctly on one try – they are LISTings with the ? changed to PRINT, and NEW was pasted in place at the top. Don't let typing get you down. It's frustrating to get to the end of a complicated multiple statement line and then notice a mistake at the beginning. My frequent errors –

(1) Forgetting the : that separates statements in a multiple statement line. I am especially prone to forgetting the colon after a PRINT statement that ends with ; or , (the technique used to make the next *PRINT statement output appear on the same display or printer line as the* last *PRINT statement output).*

(2) Forgetting the ? for PRINT, especially in direct mode, or when the PRINT statement is in the middle of a multiple statement line.

(3) Using the SHIFT key when I shouldn't, or not using it when I should. Note that using SHIFT and typing a letter key will usually display and PRINT that letter all right, but it might not RUN, giving you a syntax error—an invisible error! To cure the problem, retype the line and don't use SHIFT with the letter keys, except for special symbols.

(4) Leaving off the quotation marks at the end (or in the middle) of a long program line, for example, in a series of PRINT statements such as instructions for a game, where the PRINT statement line doesn't end at the end of a sentence. You have to type the whole line over again, just to put the quotation marks in, unless your system has a good program line editor and you take this opportunity to become familiar with it. (Actually, some BASICs will be tolerant if you leave off the final quotation mark at the end of a PRINT statement string.)



ON 1 CONDITION

First warning: if a condition is found to be false, the computer goes on to the next *line numbered statement.* That means that if you use an IF ... THEN statement in a multiple statement line, the computer will not see or execute any other statements following the IF ... THEN statement *if the condition is false*. However, if the condition is true, then the computer will take the action specified following THEN, *and* execute the rest of the statements on the same multiple statement line. As you will see, this fact is actually quite useful.

We told you that there is a whole family of IF...THEN statements. The *condition* (the part between IF and THEN) comes in six different delicious flavors, which you will see in the box on the next page. All of the members of the IF...THEN family of statements can use any one of these six different conditions between IF and THEN. But that isn't what tells the family members apart. The distinction is in what comes after THEN, that is, what action to take *if the condition is true.*

The original IF ... THEN statement, the one that is the most common statement to use for conditional branching, is like having GOTO (line number) after THEN. In fact, you *have* to enter it like this in Atari's original BASIC:

IF (condition) THEN GOTO (line number)

But even in most old-fashioned BASICs you can write it like this, *without* the GOTO after THEN (like leaving out LET in a statement to assign a value or string to a variable, you know . . .).

IF (condition) THEN (line number)

The GOTO is understood, so you don't have to put it in.

There are two possible outcomes or possible results of the *comparison.* The comparison can be *true*, or it can be *false*. Let's look at these two cases: comparison true, or comparison false.

true

IF (condition true) THEN (action to take)

Here is where the computer compares and decides that the condition is true.

The computer does what this part tells it to do when the condition is true.

false

IF (condition false) THEN (action to take)

But if the computer compares and decides that this part is not true (which means it is *false*, of course) ...

... then the computer forgets about this part and goes on to the next line numbered statement in the program.

READ

The multiple statements in line 20 take advantage of the fact that when an IF...THEN or an IF...THEN PRINT condition is *false*, it causes the computer to skip on down to the *next line numbered statement*. That means the computer ignores any statements following the false IF...THEN condition *if they are on the same line*. But if the IF...THEN condition is *true*, the rest of the IF...THEN statement is executed, as well as any statements that follow. Got that? It's another trick to using multiple statements for your future programming needs.

If the condition is *true*, PRINT all this, and go on to the next statement on this line.

```
20 IF X<0 THEN PRINT "CONDITION TRUE:"; X; "< 0" : GOTO 10
```

GOTO Line 10 and continue RUNing the program.

If the condition is *false*, go on to the next line numbered statement, and forget the rest of this line.

For most keyboards:

Use and together for "less than" <

Use SHIFT and together for "greater than" >

Use and together for "equal to" =

<= >= <>

NEW

DO IT

```
5 REM-COMPARE AND DECIDE IF X IS LESS THAN ZERO
10 READ X
20 IF X<0 THEN PRINT "CONDITION TRUE:"; X; "< 0" : GOTO 10
30 PRINT "CONDITION FALSE:"; X; "IS NOT < 0" : GOTO 10
40 DATA 4, 0, -3, 6, -2, 7, 9, -12
RUN
CONDITION FALSE: 4 IS NOT < 0
CONDITION FALSE: 0 IS NOT < 0
CONDITION TRUE:-3 < 0
CONDITION FALSE: 6 IS NOT < 0
CONDITION TRUE:-2 < 0
CONDITION FALSE: 7 IS NOT < 0
CONDITION FALSE: 9 IS NOT < 0
CONDITION TRUE:-12 < 0

?OD ERROR IN 10
```

The computer has ODed. It means Out of Data error, but don't worry about it.

And, of course, somebody's gotta be different, and it's the Northstar-style BASICs this time. If a comparison is true, these BASICs go on to the next statement, whether it is the next line numbered statement, or another statement past IF . . . THEN in the same multiple statement line. Check page 74 for the techniques that this BASIC requires, then return here.

```
5 REM-COMPARE AND DECIDE IF X IS EQUAL TO OR GREATER THAN ZERO
10 READ X
20 IF X=>0 THEN PRINT "CONDITION TRUE:"; X; "=> 0" : GOTO 10
30 PRINT "CONDITION FALSE:"; X; "IS NOT => 0" : GOTO 10
40 DATA 4, 0, -3, 6, -2, 7, 9, -12
RUN
CONDITION TRUE: 4 => 0
CONDITION TRUE: 0 => 0
CONDITION FALSE:-3 IS NOT => 0
CONDITION TRUE: 6 => 0
CONDITION FALSE:-2 IS NOT => 0
CONDITION TRUE: 7 => 0
CONDITION TRUE: 9 => 0
CONDITION FALSE:-12 IS NOT => 0

?OD ERROR IN 10
```

Same error message saying the computer tried to READ another value for X after all the values in the DATA statement had been used once. In this case, it isn't really an error. In fact, we can use it as a convenient way of stopping the computer after all the work is done.

TEST AND DECIDE

The next program has three IF...THEN PRINT statements to test the same value up to three times. There are three different conditions to test. Notice the multiple statements in lines 20, 30, and 40. When the IF... THEN PRINT condition is *false,* the computer skips to the next *line numbered statement*, and ignores any other statements following the false IF...THEN on the same line. But if the IF...THEN PRINT condition is *true*, then the rest of the IF...THEN PRINT statement is executed, *as well as any statements that follow it on the same line.*

IF ... THEN ? = IF ... THEN PRINT

```
NEW

5 REM- NEGATIVE, POSITIVE AND ZERO NUMBER TESTER
10 INPUT "INPUT 0 (ZERO) OR ANY NEGATIVE OR POSITIVE NUMBER"; N
20 IF N<0 THEN ? "YOUR NUMBER IS NEGATIVE." : ? : GOTO10
30 IF N>0 THEN ? "YOUR NUMBER IS POSITIVE." : ? : GOTO 10
40 IF N=0 THEN ? "YOUR NUMBER IS ZERO." : ? : GOTO 10
RUN
INPUT 0 (ZERO) OR ANY NEGATIVE OR POSITIVE NUMBER? 8975
YOUR NUMBER IS POSITIVE.

INPUT 0 (ZERO) OR ANY NEGATIVE OR POSITIVE NUMBER? -384
YOUR NUMBER IS NEGATIVE.

INPUT 0 (ZERO) OR ANY NEGATIVE OR POSITIVE NUMBER? 0
YOUR NUMBER IS ZERO.

INPUT 0 (ZERO) OR ANY NEGATIVE OR POSITIVE NUMBER?
```

READ

Use CONTROL/C to get out of a RUN when the program is stopped at an INPUT statement.

Let's take a closer look at one of the IF ... THEN PRINT statements.

PRINT a blank line (between loops in the RUN).

If the condition is *true*, PRINT the string, and go on to the next statement.

```
20 IF N<0 THEN ? "YOUR NUMBER IS NEGATIVE." : ? : GOTO 10
```

If the condition is *false*, go on to the next line numbered statement, and forget the rest of this line. That means don't do anything following THEN.

GOTO line 10 and continue RUNning the program.

To Print or Not To Print:
That is the Question

Now please pay close attention to these next little programs using the IF...THEN statement. Line 20 is used to decide whether to PRINT a value or not. When the condition is *false*, the computer skips on to the line 30 PRINT statement and PRINTs the value of R. But when the condition is *true*, the computer *branches* back to line 10. It does not get to line 30, and therefore it doesn't PRINT. Instead, a new value is assigned to R from the DATA. Then the new value is tested by line 20. And so on.

At first, this may seem backwards to you. Just remember that the action indicated after THEN only happens *if the condition is true.* Now go ahead and try the programs.

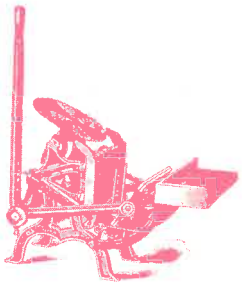

```
NEW

5 REM-DON'T PRINT NUMBERS LESS THAN ZERO
10 READ R
20 IF R<0 THEN 10
30 ? "R ="; R
40 GOTO 10
50 DATA 4, 0, -3, 6, -2, 7, 9, -12
RUN
R = 4
R = 0
R = 6
R = 7
R = 9

?OD ERROR IN 10
```

CHECK THE TRACE ON THE NEXT PAGE

The computer has ODed again. Still not to worry...

Now change the condition by replacing Line 20. (Leave out the REM if you want to.)

```
5 REM-DON'T PRINT NUMBERS GREATER THAN ZERO
20 IF R>0 THEN 10
RUN
R = 0
R =-3
R =-2
R =-12

?OD ERROR IN 10
```



Same error message saying the computer tried to READ a value for R after all the values in the DATA statement had been used once.

The computer can be instructed to READ the same data from DATA statements more than once (provided that the Out Of Data error condition has not stopped the RUN). Use the RESTORE statement like this:

120 RESTORE

It tells the computer to begin READing data again from the beginning of the first DATA statement.

stop the loop!

This is the condition

Line number → `30 IF T<10 THEN 10` → Do what this says if the condition is *true*. In this case, go to the statement with this line number.

Go on to the next line numbered statement if the condition is *false*. That means go to the next statement with the smallest line number greater than 30.

Now enter and RUN this program to stop a loop after a certain number of times, using the IF...THEN statement to do the checking.

```
NEW

5 REM-SELF-STOPPING COUNTING PROGRAM
10 ? T
20 T=T+1
30 IF T<10 THEN 10
40 ? "NOW T ="; T; "SO I STOPPED MYSELF."
RUN
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
NOW T = 10 SO I STOPPED MYSELF.
```

Our IF statement (Line 30 above) asks the profound question, "Is the value of T less than 10?" When the answer is "true," THEN 10 tells the computer to branch or loop back to Line 10. But when the answer to the profound question is "no" (when the value of T finally reaches 10), the computer ignores the rest of the IF statement and "falls through" to the next line in the program (Line 40).

AND and/or OR in IF...THEN

The double-threat actions of IF . . . THEN . . . ELSE have a counter-part on the comparison side of an IF . . . THEN statement. The logical AND and logical OR allow *more than one comparison* in the section between IF and THEN in a conditional branching statement. Here is the format.

(line no.) IF (comparison) AND (comparison) THEN (action)

Two separate comparisons, joined by logical AND, and *both comparisons must be true* for the action after THEN to be executed.

```
10 IF B >= 1 AND B <= 10 THEN ? "FROM 1 TO 10"
```

If both of these comparisons are true . . .

. . . then do this.

An application of this IF . . . AND . . . THEN format is on page 123.

(line no.) IF (comparison) OR (comparison) THEN (action)

Two separate comparisons, joined by logical OR, and if *either one is true*, the action after THEN will be executed. Also, if *both comparisons are true*, the action after THEN is executed.

```
10 IF M < 1 OR M > 10 THEN ? "NOT FROM 1 TO 10"
```

Each comparison must be complete. Two values (or numeric variables assigned values) or two strings (or string variables) must be specified for each comparison to be made, even if the same variables are used in the two different comparisons joined by AND or OR. Check the examples above.

```
10 IF M < 1 OR > 10 THEN ? "NOT FROM 1 TO 10"
```

Complete comparison.

Incomplete comparison: it doesn't tell what 10 is compared to.

Wanna get fancy? You can string a bunch of comparisons together with AND and/or OR. For example:

(line no.) IF (comparison) AND (comparison) OR (comparison) THEN (action)

If both of these are true . . .

. . . or if this is true . . .

. . . then take action!

But watch out: you must be very, very careful of your logic when getting this sophisticated . . . and complicated! Stay tuned for more exciting decisions with IF-THEN later in this broadcast. (An example of this IF . . . AND . . . OR . . . THEN format is demonstrated on page 128.)

>>>

de-loopdeloop

Flags and Stop Signs for INPUT Entry Loops

Assigning values or strings to variables is often done using the INPUT statement in a GOTO loop. This technique requires a way to inform the computer that you have finished entering data (making entries), and that the rest of the program beyond the GOTO loop is to be executed. The method uses a code word or value, also called a flag (as in "flagging down the bus"). An IF . . . THEN statement checks each INPUT variable assignment (every entry) to see if the flag or code word was entered. If a match for the code or flag is found, the IF . . . THEN statement causes the computer to branch out of the GOTO loop containing the INPUT statement(s).

```
40  IF H =  - 99 THEN 80
```

The flag checker
(not the checker flag, Indy fans.)

Forcing people to use CONTROL/C to end a RUN of your "real life" programs is definitely *gauche*. More tasteful programming etiquette requires giving the people using your program the flag option, or, for string entries, one like this:

```
10 ? "WHEN FINISHED, TYPE 'STOP'"
20 INPUT "YOUR ENTRY? "; E$
30 IF E$ = "STOP" THEN END
```


```
NEW

5  REM -- FLAG DETECTED BY IF...THEN
10  PRINT "WHEN FINISHED ENTERING DATA, ENTER -99"
20  PRINT "AND I WILL CALCULATE THE AVERAGE."
30  INPUT "HEIGHT IN INCHES?";H
40  IF H =  - 99 THEN 80
50  LET T = T + 1
60  LET C = C + H
70  GOTO 30
80  PRINT "AVERAGE HEIGHT IS ";C / T
```

flag

Line 50 keeps track of the *number of entries.*
Line 60 keeps a *running total* of all data entered.

```
RUN
WHEN FINISHED ENTERING DATA, ENTER -99
AND I WILL CALCULATE THE AVERAGE.
HEIGHT IN INCHES?40
HEIGHT IN INCHES?50
HEIGHT IN INCHES?46
HEIGHT IN INCHES?48
HEIGHT IN INCHES?45
HEIGHT IN INCHES?52
HEIGHT IN INCHES?41
HEIGHT IN INCHES?44
HEIGHT IN INCHES?45
HEIGHT IN INCHES?-99
AVERAGE HEIGHT IS 45.6666667
```

The flag

5. In this era of energy shortage and water drought, we on the West coast are becoming extremely conscious of energy consumption and would like to share our ideas with you (your turn is coming!). In terms of water, we know the facts and figures and many families measure water in terms of flushes per day and things like that. Below is a table of average consumption of various activities. Your exact figures will likely be a bit different and we encourage you to change the table to suit you. Use the table and your handy computer to write a program to *compute the average amount of water used by your family per month* in cubic feet and in gallons of water. (1 cubic ft. = 7.5 gal.) Use INPUT statements to enter the variables, and DATA statements to contain the constants (from the table).

   This may be a long program, but it really isn't that difficult. Your output statements should look like this:

```
PRINT"YOU USE APPROX. "; T;"GALLONS / MONTH OR ";T/7.5; "CUBIC FEET"
PRINT"THAT IS AN AVERAGE OF ";T/30;"GALLONS PER DAY"
```

shower.....6 gallons/minute	tub bath....20 gallons
dishwashing.....15 gallons	dishwasher..16 gallons
toilet flush.....6 gallons	(check yours)
outdoor watering...average hose	washing
10 gal./minute	machine....35 gallons
various.....you guess/day	

```
300 DATA 6,20,15,16,6,35,10
```

   Once the program works, you can change data (reduce flushes, showers, etc.) and see how much water you can save.

6. Compute your water bill!
   You attempted to compute your water usage for the month in gallons. Now for the bad news, the bill.

   Assume everyone pays a meter fee of $2.85 per month and that water is billed by the cubic foot (remember: 7.5 gallons per cubic foot).

   Write a program to compute your water bill if you are charged $0.50 per 100 cubic feet used. Enter usage in gallons (don't forget to add the meter fee). For 6000 gallons the answer is $6.85. 10,000 gallons = $9.52 and 20,000 gallons = $16.18.

7. Using the data from Problem 5 above, let's assume the water company starts to penalize large users by charging more for water used in excess of some 'normal' amount. Let's assume a family of four is allowed 8000 gallons per month and is charged $0.50 per 100 cubic feet for those 8000 gallons (plus the meter fee). For any gallonage over 8000, the charge is $1.00 per 100 cubic feet. Write a program to compute the water bill under this system (it could happen . . .).

8. Sell computer time to a neighbor . . . for fun and profit! You'll need a timeclock of some sort and a computer program to do the calculations. Charge $2.00 just for the privilege of using the system plus $0.05 per minute for the first 45 minutes and $0.03 per 100 minutes used.

   45 x .05 = 2.25
   (100 – 45) x .03 = 1.65
   connect fee = 2.00

   Total = $5.90

   Tip: What if the user uses less than 45 minutes. Don't forget to 'program' for that possibility.

9. String comparisons: You have a huge customer list of 4000 names contained in DATA statements with names and zip coded expressions in two different strings. (Why did we place zip codes in string variables instead of numbers?)

   Sample DATA statement:

```
40 DATA MARCUS,94025,LINDA, 94061, JERRY, 94061,LARRY,06542
```

   You are about to travel to zip code area 94061. Read through the names in your DATA statements and select and print only those names in zip area 94061 so you will know whom to call on while you are in the area.

Use direct mode to experiment with the SQR(X) function. Try the examples below *and* try your own ideas.


```
PRINT SQR(100)
 10
```

SQR( ) — A variable, a value, or an expression to calculate.

What goes here inside the parentheses is called the *argument* of the function— but let's not squabble.

```
? SQR(85)
 9.21955
```

```
? SQR(1), SQR(2), SQR(3), SQR(4)
 1             1.41421        1.73205        2
```

```
? SQR(5); SQR(6); SQR(7); SQR(8); SQR(9); SQR(10)
 2.23607  2.44949  2.64575  2.82843  3  3.16228
```

Now, still using direct mode, try to make BASIC find the SQR of a negative number.

```
? SQR(-25)

?FC ERROR

? SQR(100)
 10

? SQR(-100)

?FC ERROR
```

See, I told you so, BASIC tells you that you are misusing a FunCtion.

"FC ERROR" means "Function Error"

BASIC feels abused when you misuse a function. The error messages differ from one version of BASIC to another, but apparently BASIC doesn't like to compute square roots of negative numbers.

Next, try some calculations that include the SQR function.

```
? 5*SQR(9)
 15

? 5*(10+SQR(9))
 65
```

Don't forget to keep your parentheses paired, a right parenthesis for each left parenthesis, or BASIC will lay a syntax error message on you.

```
X=7 : Z=SQR(X) : ? Z
 2.64575
```

INT( )

The argument can be a variable, a value, or an expression to calculate.

INT(X) tells the computer to find the *integer* part of a positive number. The *integer* part of a number is the whole number part without the decimal fraction, like this:

INT(5.6) = 5 The integer part of 5.6 is 5.

Find the integer part of the numbers in the INT parentheses, using direct mode.


```
? INT(3.1)
 3

? INT(3.9999), INT(101.1), INT(3.14159)
 3              101          3
```

Now try negative numbers. The INT function returns the next more negative integer. INT(–5.6) = –6. The integer part of –5.6 is –6.

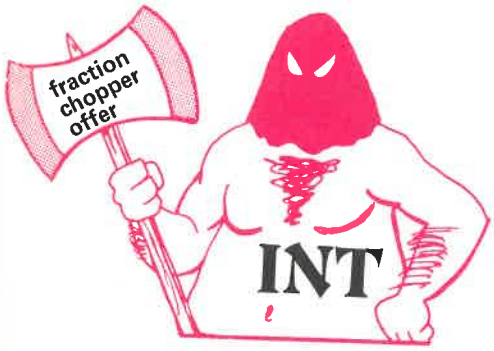
```
? INT(5.1), INT(-5.1), INT(5.9999), INT(-5.9999)
 5           -6          5           -6
```

See the difference in the way the INT works with a negative number as compared to a positive number?

Find the integer part of the square root of 52.

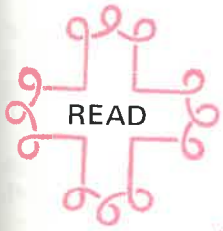
```
? SQR(52), INT(SQR(52))
 7.2111     7
```

Note the clever use of a function inside the parentheses of another function. Again BASIC starts on the inner most set of parentheses and works out. Don't forget to match each left parentheses with a right parentheses.



Now let's have the computer do the problem. Note that we instruct the computer to PRINT the value of M after each step of the rounding off process, just to help us see how the process works step-by-step.


```
NEW

10 M=17.95*.06 : ? M
20 M=M*100 : ? M
30 M=M+.5 : ? M
40 M=INT(M) : ? M
50 M=M/100 : ? M
RUN
 1.077      <- The calculated value to be rounded off.
 107.7      <- The decimal place moved two places to the right.
 108.2      <- The round off constant (.5) added on.
 108        <- The decimal fraction chopped off.
 1.08       <- The decimal point returned to its rightful place,
               two places to the left, giving us even money.
```



The same results may be accomplished in one statement. (You can use direct mode seeing as it is a one line instruction.)

```
? INT(17.95*.06*100+.5)/100
 1.08
```

1. The calculation of tax
2. Move the decimal point over 2 places to the right.
3. Add round-off constant
4. Take the INTeger part of the result of the calculations inside the parentheses.
5. Last step is to divide by 100 to move the decimal 2 places to the left, where it should be.

Remember this example to use in your own computer programs. Use INT for rounding off calculations involving dollars and cents —

```
NEW

10 M=17.95*.06
20 ? INT(M*100+.5)/100
RUN
 1.08
```

M can be the result of calculations, or it can be a value calculated earlier in the program.

The third function we want you to learn is the RND( ) function. *Random* means that something happens just by chance. It's like picking numbers out of a hat. The RND function is tricky. When you instruct the computer with the RND function, it gives you a *random* number (that looks like a decimal fraction) between zero and one. Never 0, never 1, always *between*, my friend. If the RND number (the decimal fraction) is very small, that is, very close to zero, it may be printed in floating point notation.

( ) The number, variable, or calculation found in the parentheses of these functions is often referred to as the expression, parameter, or *argument* for the function . . . wanna fight about it? There are three types of *arguments* that may be used in the parentheses for the RND ( ) function, for three different effects in most Microsoft-style BASICs. *If your BASIC doesn't seem to follow our patterns in the RUNs, see page 89.*

You could do the CONTROL/C operation to stop the loop *immediately* after you type RUN and hit RETURN, or use the Press RETURN To Continue technique (page 50). Change GOTO 20 to GOTO 15, and insert this statement: 15 INPUT ""; R$

DO IT

RND(1)

RND argument is positive.

```
NEW

10 X=1
20 ? RND(X) : GOTO 20
RUN
 .50438
 .0267824
 .388094
 .569123
 .720021
 .209046
 .599886

BREAK IN 20
```

To change the display pattern, insert a comma or semicolon before the colon.

Don't expect the RND numbers for your RUN to be the same as these. After all, they are supposed to be "random."

This happened because we pressed CONTROL/C.

Now RUN the program again, and compare the RND numbers from this RUN with the RND numbers from the first RUN.

```
RUN
 .744055
 .460434
 .433291
 .27376
 .701146
 .590584
 .457448

BREAK IN 20
```

Note that the RND numbers from this RUN of the same program are different.

Now make a written note of the *last* RND number displayed by *your* computer's RUN: _____ and then go on to the next column of activities.

RND(0)

Zero for the RND argument.

Now replace line 10 with this new line 10, and after a few sample RND numbers, stop with CONTROL/C.

```
10 X=0
RUN
 .457448
 .457448
 .457448
 .457448
 .457448
 .457448
 .457448

BREAK IN 20
```

Note this RND number, then look back at the last RND number in the previous RUN. Interesting, eh?

Now RUN again, then stop the execution with CONTROL/C and compare with the previous RUNs.

```
RUN
 .457448
 .457448
 .457448

BREAK IN 20
```

RND(-1)

RND argument is negative.

Replace line 10 again, like this:

```
10 X=-.4
RUN
 .803906
 .803906
 .803906

BREAK IN 20
```

Again, stop with CONTROL/C, do another RUN, and compare it with the others.

```
RUN
 .803906
 .803906

BREAK IN 20
```

Another negative value for X.

```
10 X=-1
RUN
 7.65943E-06
 7.65943E-06
 7.65943E-06
 7.65943E-06

BREAK IN 20
```

A "small" RND number (close to zero) is printed in floating point notation.

And yet another negative value.

```
10 X=-.3
RUN
 .905996
 .905996

BREAK IN 20
```

Hmmm...confused? Perplexed? Read on & all will be revealed.

RND ARGUMENT IS NEGATIVE

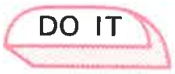
Using a negative number as the RND argument also gives you a repeated RND number each time the same RND statement is executed. However, for each different negative argument, you get a different repeated RND number. You can use decimal fractions as the negative arguments, too, such as RND(–.3) or RND(–9.32). Everytime you RUN the program or execute a statement with the *same* negative argument, you get the same RND number.

Demonstrate how RND works with negative arguments, using direct mode.

```
? RND(1); RND(-.4); RND(-.4); RND(-.4); RND(-17.5); RND(-990)
 .905996   .803906   .803906   .803906   8.37562E-06   1.96788E-03
```

Try you own demo with your own choice of RND arguments.

In some cases, it is useful to have the same list of RND numbers used in a program every time that program is RUN. This might be the case for some *simulation* programs. (You'll hear more about simulations later on.) In effect, the RND(–.3) gives the computer an initial RND number to start off the list, and the RND(1) picks up from there and keeps generating more RND numbers. Try our list repeater program.




```
NEW

5 REM-RND NUMBER LIST REPEATER
10 X=RND(-.3)
20 ? RND(1)
30 GOTO 20
RUN
 .905996
 .270418
 .504185
 .885868
 .22101
 .457995
```

Change the display pattern with a comma or semicolon.

Use CONTROL/C to stop the RUN.

```
BREAK IN 20

RUN
 .905996
 .270418
 .504185
 .885868
 .22101
 .457995

BREAK IN 20
```

New RUN, same list.

If your version of BASIC didn't seem to follow the format of our RUNs, not to worry. In some BASICs, you get repeated new RND numbers between 0 and 1 only if the argument is one. Arguments greater than one—say, RND(10)—produce random digits from 1 to 10 inclusive. Try it out! Atari, DEC BASIC Plus and some others require a separate program statement to tell the computer to generate a different RND list of numbers for each RUN, and the statement looks like this:

```
6 RANDOMIZE
```

or like this:

```
6 RANDOM
```

The RANDOM or RANDOMIZE statement comes before the first RND function in the program.

NOW IS THE TIME FOR ALL GOOD PEOPLE TO *EXPERIMENT*

How about if we want RND numbers from 1 to 10 instead of from 0 to 9.? Simple - just add 1 to the value of X. This could be done anytime after the RND number is generated. Modify the program RND INTEGERS EXPLAINED like this, with the X=X*10 column removed, and X=X+1 added:

Take out line 30.

```
30
10 ? " X=RND(1)", " X=INT(X)", " X=X+1"
40 X=INT(X) : ? X,
45 X=X+1 : ? X
```

You do a LISTing to see the modified program. It should RUN like this:

```
RUN
 X=RND(1)        X=INT(X)        X=X+1
 .951037         9               10
 .376529         3               4
 .220719         2               3
 .379046         3               4
 .818024         8               9
 .989366         9               10
5.23684E-03      0               1
 .281303         2               3

BREAK IN 45
```

Or get real fancy and do it in one line like this:

```
NEW

10 ? INT(10*RND(1))+1; : GOTO 10
```

Keep your parentheses paired!

Now why do you suppose that semicolon is there?

```
RUN
 9  1  8  1  2  5  7  5  1  2  6  6  5  9  2  8  10  5  8  2  8  9  6
 7  1  6  7  6  9  10  3  7  8  5  6  1  8  10
BREAK IN 10

RUN
```

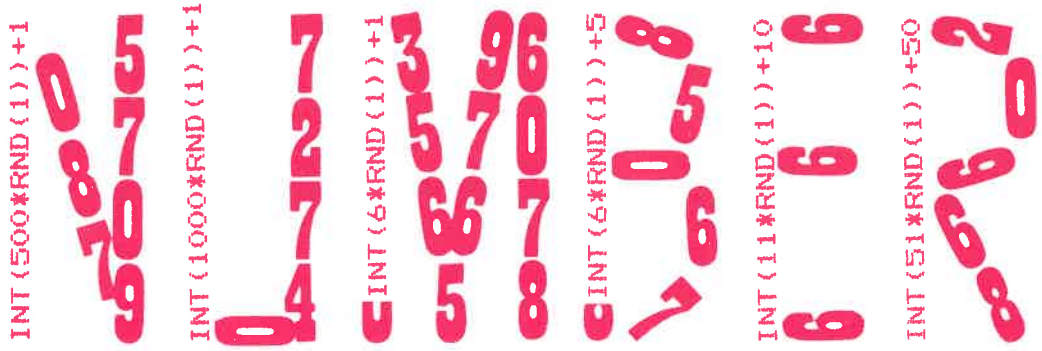
New RUN, new list of RND integers between 1 and 10.

```
 4  7  4  2  7  5  1  4  7  5  2  6  5  10  1  4  5  9  3  5  3  8  9
 10  9  3  7  9  2  4  7  7  1  8  9  2  8
BREAK IN 10
```

How about random numbers from 1 to 100? Simple!

```
10 ? INT(100*RND(1))+1; : GOTO 10
RUN
 65  42  76  57  63  5  57  27  7  22  52  88  9  60  83  42  90  32
 83  98  44  62  11  11  64  80  90  54  31  72  91  45  14  86  85  81
 35  80  90  15  95  71  47  93  74  82  19  96  8  93  83  79  85
BREAK IN 10
```

Go ahead and try it for numbers from 1 to 500 inclusive, and 1 to 1000. Feeling sharp? Then how about random numbers from 1 to 6? Even trickier, from 5 to 10, from 10 to 20, and from 50 to 100. Answers are hidden two pages forward.



EXPLAINED

Don't forget that (1) you can use ? for PRINT when typing in instructions, and (2) a PRINT with no instruction after it leaves a blank line in the printout.

```
310 X=INT(100*RND(1))+1
```

Line 310 gets a random number between 1 and 100 and assigns it to variable X.

```
420 IF G=X THEN ? "YOU GOT IT!!! LET'S PLAY AGAIN " : GOTO 310
```

Line 420 compares the computer's number (value of X) with the player's Guess (value of G), and if they are the same (condition *true*), the computer tells you "YOU GOT IT!"

```
430 IF G<X THEN ? "TOO SMALL. GUESS AGAIN." : GOTO 410
```

Line 430 prints TOO SMALL if the Guess G is smaller than X.

```
440 ? "TOO BIG. GUESS AGAIN." : GOTO 410
```

Line 440 is only executed if the IF conditions in Lines 420 and 430 are *false* and the computer "falls through" to Line 440.

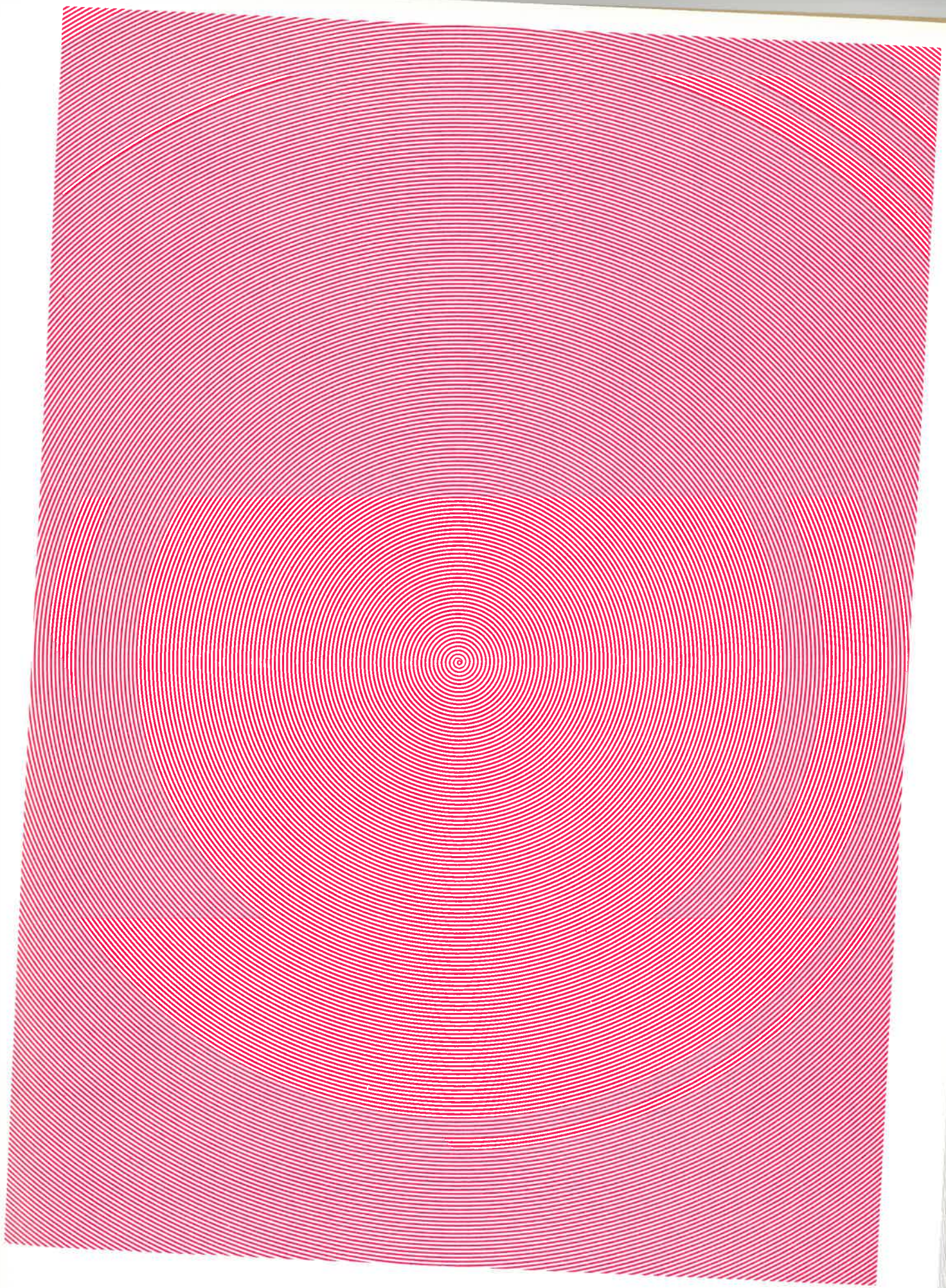
Line 440 prints TOO BIG, because if the guess is not equal to or smaller than the computer's number X, then it must be bigger. The computer doesn't need an IF statement to decide that!

Do you understand how the program works? Then go ahead and enter it and RUN it. Hope you get it all typed in right the first time you try. If you don't and it doesn't RUN, get a LISTing of the program and look for your error(s).

Extra for experts: Develop a version of this game for two players who alternate their guesses until one of them gets the number.

Here are clues for Northstar-style BASIC users to adapt the program to the IF . . . THEN . . . ELSE format.

```
420 IF G = X THEN ? "YOU GOT IT!!! LET'S PLAY AGAIN." ELSE 430
421 GOTO 310
430 IF G <= X THEN ? "TOO SMALL. GUESS AGAIN." ELSE 440
431 GOTO 410
```



Introducing FOR-NEXT


```
NEW

5 REM-USING THE FOR-NEXT LOOP CONTROL VARIABLE TO COUNT OFF LOOPS
10 FOR F=1 TO 7
20 ? "F =";  F
30 NEXT F
40 ? "NOW F =";  F
RUN
F =  1
F =  2
F =  3
F =  4
F =  5
F =  6
F =  7
NOW F =  8
```

This is a FOR statement.

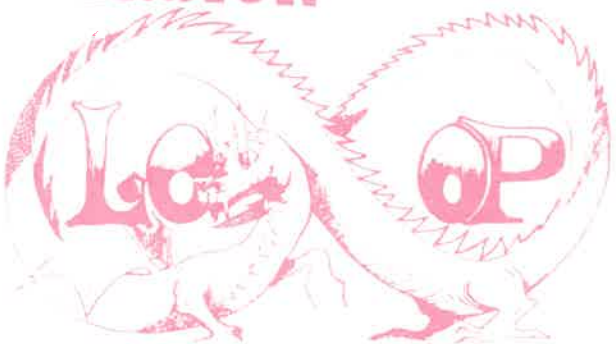
This is a NEXT statement.

FIRST VERSION

Now do the following program, and compare it to the one you just tried. Lines 10, 20 and 30 of the first version are all in one multiple statement line, Line 10 in the second version.

```
NEW

10 FOR F=1 TO 7 : ? "F =";  F : NEXT F
20 ? "NOW F =";  F
RUN
F =  1
F =  2
F =  3
F =  4
F =  5
F =  6
F =  7
NOW F =  8
```

SECOND VERSION



Practice makes purfect

DO IT

Here are some practice programs using FOR-NEXT loops.


```
NEW

10 A=5 : B=10
20 FOR C=A TO B : ? "C =";  C : NEXT C
RUN
C =  5
C =  6
C =  7
C =  8
C =  9
C =  10
```

Replace Line 10 with the one below.

```
10 INPUT "A =";  A : INPUT "AND B =";  B
```

And add this Line 30.

```
30 ? : GOTO 10
LIST

10 INPUT "A =";  A : INPUT "AND B =";  B
20 FOR C=A TO B : PRINT "C =";  C : NEXT C
30 PRINT : GOTO 10

RUN
A =? 4
AND B =? 8
C =  4
C =  5
C =  6
C =  7
C =  8

A =?
```

Try these values, but don't stop there. Get the hint?

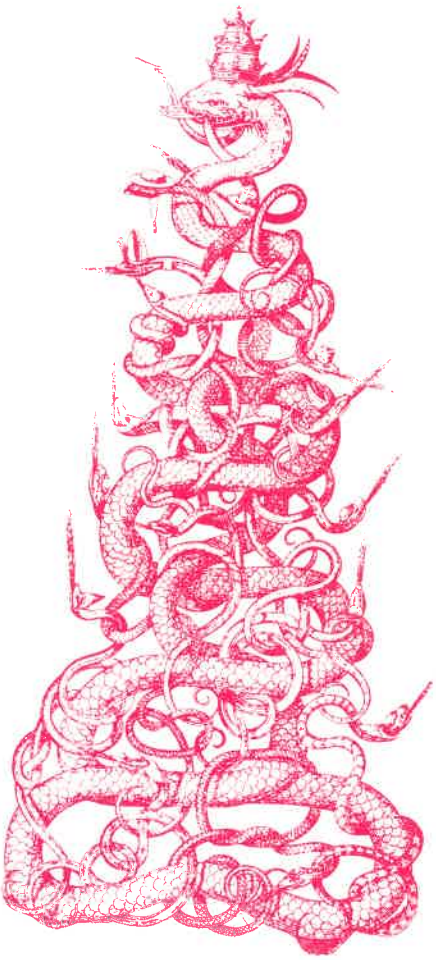
A=0, B=6
A=–9, B=2
A=3.3, B=7.5
A=6, B=3
etc., etc., etc.....

TILT!

Try these input values and what ever else you want to try.

A=1, B=10, STEP=2
A=1, B=6, STEP=.7
A=6, B=1, STEP=−1.5
etcetera, etctera, etcetera...

```
NEW


5 REM-FOR STATEMENT WITH STEP
10 INPUT "A =";  A : INPUT "B =";  B : INPUT "STEP";  S
20 FOR C=A TO B STEP S : ? "C =";  C : NEXT C
30 ? "OUT OF THE LOOP BECAUSE C =";  C : GOTO 10
RUN
A =? 1
B =? 10
STEP? 2
C =  1
C =  3
C =  5
C =  7
C =  9
OUT OF THE LOOP BECAUSE C =  11
A =? 1
B =? 6
STEP? .7
C =  1
C =  1.7
C =  2.4
C =  3.1
C =  3.8
C =  4.5
C =  5.2
C =  5.9
OUT OF THE LOOP BECAUSE C =  6.6
A =? 6
B =? 1
STEP? -1.5
C =  6
C =  4.5
C =  3
C =  1.5
OUT OF THE LOOP BECAUSE C =  0
A =?
```



We stopped here, using CONTROL/C and RETURN to get out of a RUN waiting at an INPUT statement, *but we sure hope you didn't!!!* Go ahead and *EXPERIMENT!*

FOR~NEXT in direct mode

Note that you can put a FOR–NEXT statement and the body or working section that comes between the FOR and NEXT statements, in a one line multiple statement. That means you can use FOR–NEXT in one line *direct mode* instructions. Of course, it only works if all the instructions will fit on one line. Try the ones below.

Tickle your computer's fancy with this direct mode statement to give the computer the giggles.

Make this into a line numbered program, if you wish.

```
H$="HA " : FOR C=1 TO 50 : ? H$; : NEXT C
HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA
HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA HA
HA HA
```

Now use direct mode to print a table of squares and square roots for 1 to 16.

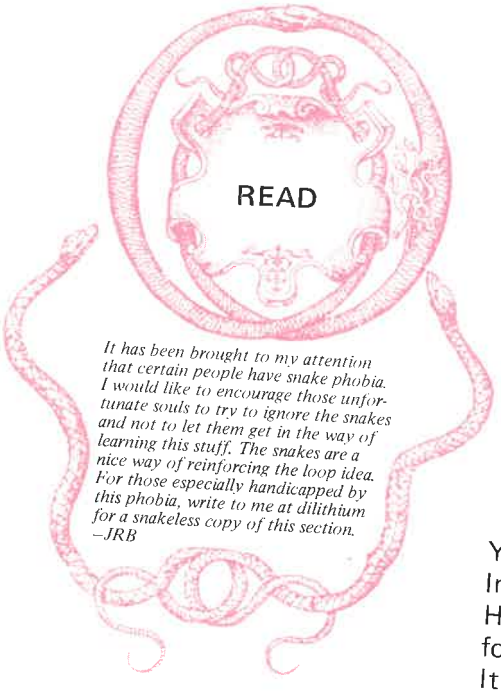
Make this a four-statement program, if you want to.

This part PRINTs the headings...

...and this part does the table. Aren't FOR–NEXT loops wonderful?

```
? " X", " X↑2", " SQR(X)" : FOR X=1 TO 16 : ? X, X↑2, SQR(X) : NEXT X
 X             X↑2           SQR(X)
 1             1             1
 2             4             1.41421
 3             9             1.73205
 4             16            2
 5             25            2.23607
 6             36            2.44949
 7             49            2.64575
 8             64            2.82843
 9             81            3
 10            100           3.16228
 11            121           3.31663
 12            144           3.4641
 13            169           3.60555
 14            196           3.74166
 15            225           3.87298
 16            256           4
```

Try your own FOR-NEXT loop ideas. Do it now!



Loops Inside of Loops

You can use one FOR ... NEXT loop *inside* another FOR ... NEXT loop. In computer jargon, they say one loop is *nested* inside the other loop. However, you can't "overlap" two loops, that is, with the NEXT statement for the first FOR coming *before* the NEXT statement for the second FOR. It's easy to understand by looking at it.

RIGHT

```
NEW

5 REM-NESTED FOR-NEXT LOOPS
10 FOR A=1 TO 3
20 FOR B=1 TO 5
30 ? "NESTED LOOPS"
40 NEXT B
50 NEXT A
RUN
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
NESTED LOOPS
```

This loop is nested inside this loop

wrong

```
5 REM-CROSSED FOR-NEXT LOOPS
10 FOR A=1 TO 3
20 FOR B=1 TO 5
30 PRINT "CROSSED LOOPS"
40 NEXT A
50 NEXT B
RUN
CROSSED LOOPS
CROSSED LOOPS
CROSSED LOOPS

?NF ERROR IN 50
```

These loops overlap, and therefore are not nested one inside the other.

Your error message may be different. NF ERROR means Next without For. This means the computer came to a NEXT statement, but since the loops were crossed, it could not find the FOR statement with the same control variable. Notice that the computer did execute the first FOR—NEXT loop with the control variable A. But then the computer just followed the rules for the way FOR—NEXT loops operate. After the third time through the loop, A=4, so the computer went on to the next statement after 40 NEXT A. That means it came to 50 NEXT B directly from the A loop. The computer thought it had found a NEXT statement (NEXT B) without a FOR statement with the same control variable. That's why we got an error message. Now if a smart person like you is confused, imagine what happens to a dumb machine like the computer.

And now, here's two loops nested inside another loop, and yet another loop outside.

```
NEW

10  FOR A = 1 TO 8
20  FOR B = 1 TO 100: PRINT "HA ";: NEXT B
30  FOR C = 1 TO 100: PRINT "HEE ";: NEXT C
40  NEXT A
50  FOR D = 1 TO 200: PRINT " ";: NEXT D
60  PRINT "COUGH CHOKE"

RUN
```

Check this PRINT statement that prints 200 blank spaces. While humorous here, the technique is handy.

There is an alternative to the Press RETURN To Continue technique for controlling the length of time an output stays on the display screen. Automatic time delays for continuing and/or fast changing video screen displays (often the result of a program GOTO or FOR-NEXT loop) can also be accomplished using an "empty" FOR-NEXT loop. The number of iterations (loops) controls the length of the delay before the next output is displayed. (Some BASICs have special instructions for this purpose, too.)

```
NEW
10 FOR J = 1 TO 5
20 ? "START TIMING NOW."
30 FOR K = 1 TO 500
40 NEXT K
50 ? "STOP!"
60 NEXT J
RUN
```

Change this value to change delay time.

TIMING CHART

100	500	2000	10000	Your Test Values

FOR-NEXT

Executes a controlled loop between the FOR and the NEXT statements for a specified number of times.

(line no.) FOR (variable) = (parameter) TO (parameter) STEP (parameter)
(line no.) NEXT (variable)

```
10 FOR X = 1 TO 30
20 PRINT X,
30 NEXT X
40 END
```

1 is initial value of X, 30 is final value of X, no STEP is specified, therefore implied to be one (1). It will print values 1 – 30 and then continue to next statement (END).

```
10 FOR 2 = N TO M
```

N, M must have been previously assigned values.

```
10 FOR Z = 1 TO 2 STEP .1
20 PRINT Z,
30 NEXT Z
```

Will print values from 1 to 2 in increments of .1, as shown below: 1.1, 1.2, 1.3, 1.4, 1.5, . . . 2.0

```
10 FOR X = 30 TO N STEP –1
```

Will count down (negative increments) from 30 to N.

WORD

Now, let's put three more functions through some paces, and then we'll explain in full. You can get a first clue as to what parts of a string are manipulated by these functions, just by looking at their names: LEFT$, RIGHT$, and MID$. Note how we use the value of LEN(I$) to establish the limit of the FOR-NEXT loop control variable.

Also study our clever use of the changing values of the FOR-NEXT loop's control variable (K) in the LEFT$ and RIGHT$ functions, to count out how many characters in I$ are to be PRINTed for each trip through the loop.

LEFT$( )

```
NEW

10 I$="INSTANT BASIC"
20 FOR K=1 TO LEN(I$)
30 ? LEFT$(I$,K)
40 NEXT K
RUN
I
IN
INS
INST
INSTA
INSTAN
INSTANT
INSTANT
INSTANT B
INSTANT BA
INSTANT BAS
INSTANT BASI
INSTANT BASIC
```

Note the use of control variable K to specify how many characters are PRINTed by the LEFT$ function.

RIGHT$( )

Now replace line 30 and RUN the program again. (LIST if you want to.)

```
30 ? RIGHT$(I$,K)

RUN
C
IC
SIC
ASIC
BASIC
 BASIC
T BASIC
NT BASIC
ANT BASIC
TANT BASIC
STANT BASIC
NSTANT BASIC
INSTANT BASIC
```

GAMES

With the strings assigned to X$ and Y$ still in the computer's memory, you can enter direct mode statements to show how RIGHT$ works.

```
? RIGHT$(X$,4)
6789
```

Try other values in the RIGHT$ argument.

```
? RIGHT$(Y$,4)
TERS
```

MID$ ( ) returns the part of the string identified by the numbers following the string variable inside the parentheses. The first number says where to start, and the second number tells how many characters to PRINT. If there is no second number in the MID$ argument, the computer starts at the specified character and prints that character and the rest of the string.

Example: MID$(X$, 4) says to return the part of the string starting with the 4th character, and including the rest of the string. MID$(X$, 4, 3) says to return the part of the string starting with the 4th character, but only that character and the next 2 characters (3 characters total are returned).

With the X$ string of numbers and the Y$ string of letters still in the computer's memory, use direct mode to demonstrate the two forms of the MID$ function for both strings.


```
? MID$(X$,4), MID$(X$,4,3)
456789        456

? MID$(Y$,4), MID$(Y$,4,3)
PUTERS        PUT
```

Users of Northstar, Atari, and some other BASICs will use the following substring function, that is similar (but not identical, take note) to MID$. Most BASICs using this format for identifying substrings also require DIMensioning of string variables at the beginning of the program.

PRINT X$(2,8) means print the string assigned to X$ from the 2nd character to the 8th character in the X$ string. An error will result if X$ has less characters than specified by either substring argument.

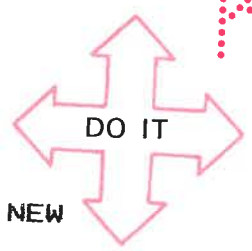
PRINT X$(2) means print the string assigned to X$ starting with the 2nd character and going all the way to the end of the string.

PRINT X$(2,2) means print only one character: in this case, only the 2nd character in the string.

PRINT X$(1,1) means print only the first character in a string.

PRINT X$(LEN(X$)) Means print only the last character in the string. The value for the LENgth of the string is also the character position number for the last character in the string. Logic: a substring that starts with the last character in a string, and going to the end of the string (which is also the last character), produces only that last character. (Try it to convince yourself.)

KEEPING SECRETS



As you can tell, ASC(X) gives the ASCII numerical value of each character. And to go the other way, use CHR$(X) to change a numerical value to its ASCII character equivalent. You can use this function to convert messages coded in ASCII numbers to their equivalent characters.

```
5  REM - CAPT. MIDNIGHT'S ELECTRONIC SECRET CODE DECODER
10  READ A
20  PRINT  CHR$ (A);: GOTO 10
30  DATA  80,82,79,71,82,65,77,77,69,82,83,32,68,79,32,73,84,32,66,65,83,
    73,67,65,76,76,89

RUN
PROGRAMMERS DO IT BASICALLY
?OUT OF DATA ERROR IN 10
```

"Forcing quotes with CHR$(34):"

```
10  PRINT  CHR$ (34);"HELLO,"; CHR$ (34);" SHE SAID."

RUN
"HELLO," SHE SAID.
```

"One Ringee Dingee..."

DO IT

Are your ears ringing? How about your computer terminal? There is a "sound off" character often called BELL, recognized by BASIC as ASCII number 7. If you use this ASCII instruction in a PRINT statement, it will cause a bell to ring on many terminals, or a "beep" on many others. CHR$(7) is a favorite with computer game programmers to signal a correct response or answer, an error, or just to add interest to the program. To ring the bell or honk the horn, try these experiments in immediate or direct mode, or convert them to short line-numbered programs.

```
? CHR$(7)
```

If you don't get a beep or bell, either your terminal is mute, or your BASIC uses a different ASCII number. Example: Atari BASIC uses CHR$(253).

No visual output, just a ringing in your ear.

```
FOR J=1 TO 8 : ? CHR$(7) : NEXT J
```

Eight bells and all is well. Also eight line spaces, one for each time the ? (PRINT) statement is performed.

Now RUN a program that *only* prints the members names, but *not* their addresses.

```
NEW
```

DIM M$ if needed.

```
5  REM   PRINT ONLY THE MEMBERS' NAMES
10  FOR K = 1 TO 5: READ M$: PRINT  LEFT$ (M$,14): NEXT K
900   DATA  J.R.BROWN        13140 FRATI LN. SEBASTOPOL     CA95472
901   DATA  PATTI MILLER     11000 SW 11TH STBEAVERTON       OR97005
902   DATA  PAUL BROWN       263 DEARBORN     IOWA CITY      IA52240
903   DATA  H. OILER         THE ASTRODOME    HOUSTON        TX77025
904   DATA  K. KONG          EMPIRE ST. BLDG.NEW YORK        NY10001
```

Oops! A mistake in entering data in the DATA statement. This could mess things up. Be sure to double-check yours.

```
RUN
J.R.BROWN
PATTI MILLER
PAUL BROWN
H. OILER
K. KONG
```

Without typing NEW, replace lines 5 and 10, and add the rest of the lines in the Address Labels program. But it's a bother to retype those DATA statements, so don't type NEW!

```
5  REM      PRINT ADDRESS LABELS
10  FOR K = 1 TO 5
20  READ M$
30  PRINT  LEFT$ (M$,14)
40  PRINT  MID$ (M$,15,16)
50  PRINT  MID$ (M$,31,15);"  "; MID$ (M$,46,2);"  "; RIGHT$ (M$,5)
60  PRINT
70  NEXT K
```

If you have added more DATA statements, change the upper limit for K accordingly.

These spaces separate city from state.

These spaces separate state from zip code.

LIST to see the expanded program, then RUN it.


```
RUN
J.R.BROWN
13140 FRATI LN.
SEBASTOPOL      C  A9  95472

PATTI MILLER
11000 SW 11TH ST
BEAVERTON            OR  97005
```

Uh-oh . . . Our error in entering line 900 shows up here.

(The RUN continues until all addresses have been printed.) Again, *without* typing NEW, add these three lines to the program. Now the program will only print mailing labels for the people in the city you specify.

DIM M$ and C$ if needed.

```
8  INPUT "ENTER CITY NAME:";C$
9  PRINT
25 IF MID$(M$,31,LEN(C$)) <> C$ THEN 70
```

If they don't match (comparison true) then skip the PRINT statements and try the next DATA item.

LIST to see the expanded program.

```
RUN
ENTER CITY NAME:NEW YORK

K. KONG
EMPIRE ST. BLDG.
NEW YORK             NY  10001
```

MID$(M$,31,LEN(C$))

See LEN used to set the MID$ function's second argument for how many characters to include in the substring. This matches the length of the "city name" substring of M$ to the length of C$, so that any position-holding spaces following the city name in the DATA statements are *excluded* from the string comparison in line 25. How clever of us to remember that extra spaces have ASCII numbers that would mess up the string comparison of C$ to the city name substring in M$.

NOSPACE

READ

Remember that many BASICs leave one leading character space in front of a *positive* value (where the plus sign could go, even if it isn't printed). When such a positive value is changed to a string with STR$, the string also includes that space. Count the characters in the output from the last program to verify this. Remember that the decimal point is also counted as a character.

Converting a value to a string gives you more control over how and where that number is printed by the computer. This is because you can apply string functions to a "converted" value.

Let's say you want to print the value of a FOR-NEXT variable for each trip through the loop. You want these numbers to print or display one after another, but your BASIC is the type that includes a leading space before positive values. Try this technique to eliminate the leading space.

DO IT

```
NEW

10 FOR K=1 TO 8 : PRINT K; : NEXT K : PRINT
20 FOR K=1 TO 8 : K$=STR$(K) : PRINT MID$(K$,2); : NEXT K
RUN
 1  2  3  4  5  6  7  8
12345678
```

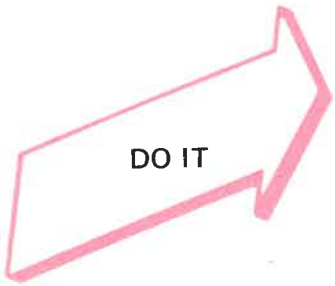
We don't want those spaces in there. So we change the value to a string, and print the string starting with the second character. That leaves out the space!

Coming soon — real applications for values converted to strings. Keep an eye out for STR$ used to change values to strings in order to control the print position of numbers.

Now let's master another interesting and useful BASIC function that gives you more ways to get what you want out of your computer. The more control you have, the better your programs will be. Each new BASIC function you learn will bring you closer to using BASIC and your computer to the fullest extent possible.

getting arty

In the following program we use the PRINT statement to draw a picture or graphic representation of the Leaning Tower of Pizza (no anchovies, please).



DO IT

```
NEW

10 ? "XXX"
20 ? " XXX"
30 ? "  XXX"
40 ? "   XXX"
50 ? "    XXX"
RUN
XXX
 XXX
  XXX
   XXX
    XXX
```

Now do it with TAB.

```
NEW

10 PRINT TAB(0); "XXX"
20 PRINT TAB(1); "XXX"
30 PRINT TAB(2); "XXX"
40 PRINT TAB(3); "XXX"
50 PRINT TAB(4); "XXX"

RUN
XXX
 XXX
  XXX
   XXX
    XXX
```

How about a more automated tower graphic? Note the use of the FOR-NEXT loop control variable to get the "lean."

```
NEW

10 FOR X=0 TO 4
20 ? TAB(X); "XXX"
30 NEXT X
RUN
XXX
 XXX
  XXX
   XXX
    XXX
```

Many brands of computers have special instructions for producing and controlling graphics. These differ so greatly from one BASIC to another that we won't even attempt to summarize them here. Effective use of graphics producing statements and functions requires the same general techniques for manipulating strings and values with assignment statements, branching statements, and loops that you have been learning. So in that respect you are ahead of the game.

Some BASICs (Radio Shack and Atari, for example) have a built-in set of graphics figures, each with its own ASCII code number, just as if they were alphanumeric characters. You can check using this program.

```
10 FOR G = 1 TO 256 : ? CHR$(G); : NEXT G
```

In general, you can turn on and off little dots (actually tiny rectangles) that are arranged in a grid (row and column) pattern, specifying their position for display by the crosspoints of row and column numbers. On systems with color graphics, you also get to choose from a set of colors, and often color intensity, too. Some allow you to predefine shapes, and to store them for later use, when you can also specify the size, position and orientation (tilt or rotation) of that predefined pattern. And then there is the sound-generating capability for producing music on some systems. Composers take note.

YES or NO Entry Tests

Here is a handy routine to use in programs such as Number, A Guessing Game (see page 92). You can add it to the beginning of that program to skip or not skip the instructions for playing Number. It is, of course, more useful for providing a choice to skip longer, more complicated or time-consuming instructions or other parts of a long program.

```
NEW

5  REM -- YES OR NO TESTER FOR NUMBER GUESSING GAME
20  INPUT "DO YOU NEED INSTRUCTIONS?";R$
30  IF RIGHT$(R$,1) <> "Y" AND RIGHT$(R$,1) <> "N" THEN PRINT :
PRINT CHR$(7); "TYPE 'Y' FOR YES OR 'N' FOR NO" : PRINT : GOTO 20
40  IF  RIGHT$ (R$,1) = "N" THEN 310
50  GOTO 210
60  REM -- PROGRAM CONTINUES

RUN
DO YOU NEED INSTRUCTIONS?MAYBE

TYPE 'Y' FOR YES OR 'N' FOR NO

DO YOU NEED INSTRUCTIONS?
```

The fence straddler gets the error message, because the first character in R$ is M, instead of Y or N.

Examine that complicated-looking line 30. It starts with an IF . . . THEN statement that has two comparisons joined by the logical AND (see page 75), followed by four more statements in the same multiple statement line. This is the Microsoft-style version, where the multiple statements following IF . . . THEN are only executed if the comparison part evaluates as true (in this case, if R$ doesn't start with a Y or an N).

You can use the bell or beeper to call attention to the error message by including CHR$(7) in a PRINT statement. The blank PRINT statements help to visually separate the error message from the INPUT prompt string.

Here is the same sort of Y or N choice and check for the end of Number, A Guessing Game. However, in this version *only* an entry of Y or N is acceptable, not YES, or NO, or a null string (no entry), or anything else. Check line 510 carefully to see why.

```
420  IF G = X THEN  PRINT "YOU GOT IT!!!": GOTO 500
500  INPUT "DO YOU WANT TO PLAY AGAIN (Y OR N)?";R$
510  IF R$ <  > "Y" AND R$ <  > "N" THEN  PRINT : PRINT  CHR$ (7);
     "TYPE 'Y' FOR YES OR 'N' FOR NO": PRINT : GOTO 500
520  IF R$ = "Y" THEN 20
530  PRINT "LET'S PLAY AGAIN SOMETIME. BYE-BYE"

RUN
YOU GOT IT!!!
DO YOU WANT TO PLAY AGAIN (Y OR N)?YES

TYPE 'Y' FOR YES OR 'N' FOR NO

DO YOU WANT TO PLAY AGAIN (Y OR N)?
```

We pressed RETURN without an entry.

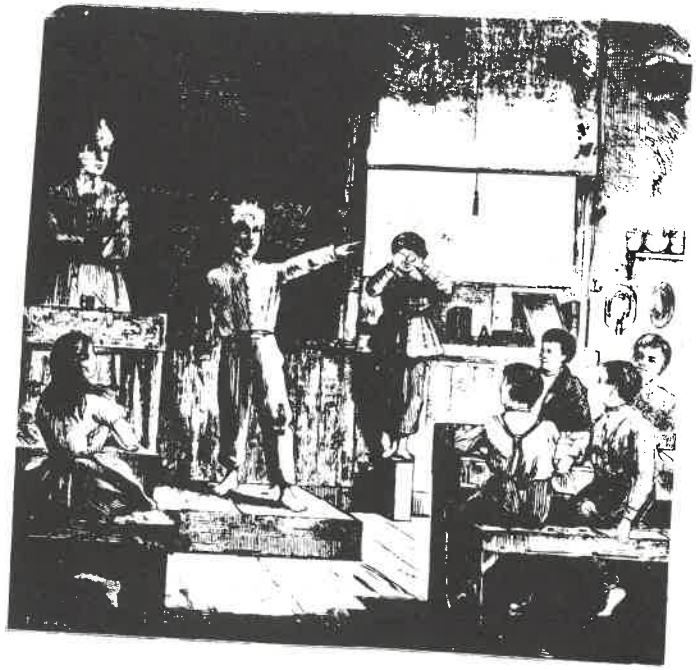
```
TYPE 'Y' FOR YES OR 'N' FOR NO

DO YOU WANT TO PLAY AGAIN (Y OR N)?N
LET'S PLAY AGAIN SOMETIME. BYE-BYE
```

revisited

Now modify (rewrite and reenter) Line 10 so that the program gives practice with the addition of numbers between 0 and 99.

Change Line 20 so that four dashes or underlines are printed under the numbers in the addition problem.


```
NEW

10 A=INT(RND(1)*100) : B=INT(RND(1)*100)
20 PRINT " "; A : PRINT "+"; B : PRINT "----" : INPUT C
30 IF C=A+B THEN PRINT "RIGHT ON!" : PRINT : GOTO 10
40 PRINT "YOU GOOFED, TRY AGAIN." : PRINT : GOTO 20

RUN
  62
+ 66
----
? 128
RIGHT ON!

  13
+ 84
----
? 97
RIGHT ON!

  77
+ 82
----
? 159
RIGHT ON!

  10
+ 4
----
?
```



But look here. This isn't the way you see problems set up in math books.

WRONG

```
  10
+ 4
----
```

RIGHT

```
  10
+  4
----
```

There are as many ways of solving a programming problem as there are of skinning cats. (My apologies to cat lovers.) We could use the TAB function to correctly place the numbers in the problems in standard form, with the 1's in the 1's place and the 10's in the 10's place, and so on for problems with more than two digits.

FUNCTIONS UNLIMITED: DEFINE YOUR OWN

You can create your own specialized functions, using the DEF FN (Define Function). This is the form in which you DEFine your FuNction for the computer.

DEFine FuNction

DEF FNA (V) = expression in which V is the "dummy variable"

dummy variable for the function argument

variable to distinguish this defined function from any others in the same program

Once you define a function in a program statement, you may use the function just as you would any other function that is part of BASIC. However, there are several tricky things about about defined functions.

(1) You must define your function in the program using a DEF statement *before* you use the function itself in other statements later in the program.

(2) DEF is *only* used in the statement where you DEFine the function, similar to the way DIM is used.

(3) The "dummy variable" in the DEFinition is just a place holder: it shows the computer where in the function the real program variable should substitute for the dummy variable.

(4) Don't be confused: the variable that identifies the function itself goes right after FN_ where we have the blank. The dummy variable for the function argument (we use V in the DEF FN_(V) statement), is where the real program variable gets "plugged in" or substituted when the function is used in the program.

(5) DEF FN_ may be used for string variables and string manipulation—but not in all versions of BASIC.

To demonstrate, let's go back to the program we used to line up decimal points. We want a defined function that will provide a value for the TAB argument. We use P (for Point) as the function ID variable (FNP).

```
10 A=1.346 : B=225.1 : C= 11.73
20 DEF FNP(V)=6-LEN(STR$(INT(V)))
30 ? TAB(FNP(A)); A
40 ? TAB(FNP(B)); B
50 ? TAB(FNP(C)); C
RUN
     1.346
   225.1
    11.73
```

Sure saved a lot of typing!

Every time the computer finds FNP, it does this operation, using the real program variable's value in place of the dummy "place-holder" variable V. The result is a value for the TAB argument in our example program.

20 DEF FNP(V)=6—LEN(STR$(INT(V)))

dummy variable

function ID (identifying) variable

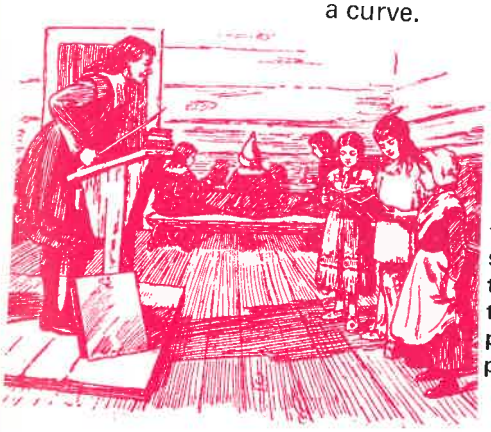
30 ? TAB(FNP(A)); A

function ID variable

The real variable, where the dummy variable was. Now the computer does to A whatever FNP was set equal to for the dummy argument V in the DEF statement.

If you understand the concept of "defining your own," write a program with a DEF FN of use to you for your own interests.

For any graph, you must decide on a scale that will be both printable by the computer and also show the characteristics of the mathematical function being graphed. Since the sine of an angle can only have values between +1 and −1, we must expand the scale in order to plot a curve that looks like a curve.



```
TAB(10*(1 + SIN(P)))
```

Multiplying by 10 gives us an expanded scale of 0 to 20 on the Y-axis, that is, 10 character spaces "above" and "below" the real X-axis. That also means there are 20 character print positions where a point may be plotted or printed by the computer.

Adding one to the value of the sine gives values between 0 and +2, instead of between −1 and +1. The reason for doing this is to avoid negative arguments in the TAB function, since the computer can't TAB to a negative character position. For us non-math types it's sort of like adding a +1 to get a RND number between 1 and 10 instead of 0 and 9 in the RND integer routine we used before.

```
FOR P=0 TO 2*3.14159 STEP .3
```

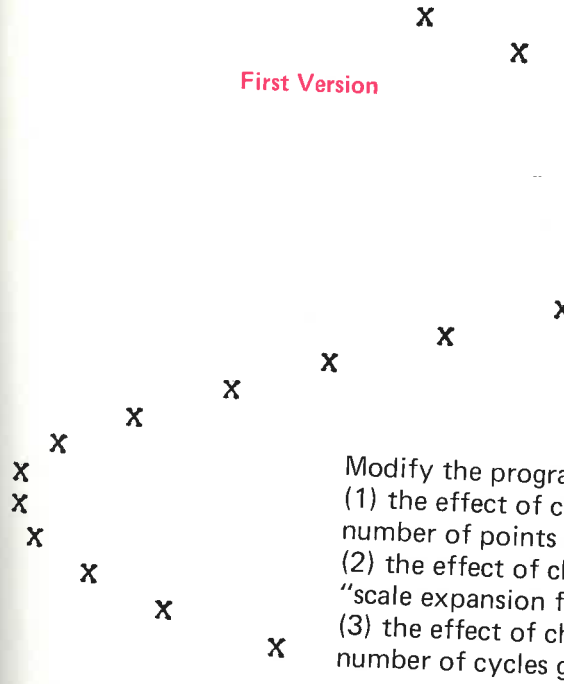
By changing this value, we can specify how many cycles will be plotted. 2*PI is one complete cycle, so 4*PI would be 2 cycles.

We can specify how many points we wish plotted along the curve by varying the STEP value. The larger the STEP value, the fewer the points that are plotted.

DO IT

```
5 REM-PLOTTING A SINE WAVE
10 FOR P=0 TO 2*3.14159 STEP .3
20 PRINT TAB(20*(1+SIN(P)));"X"
30 NEXT P

RUN
```

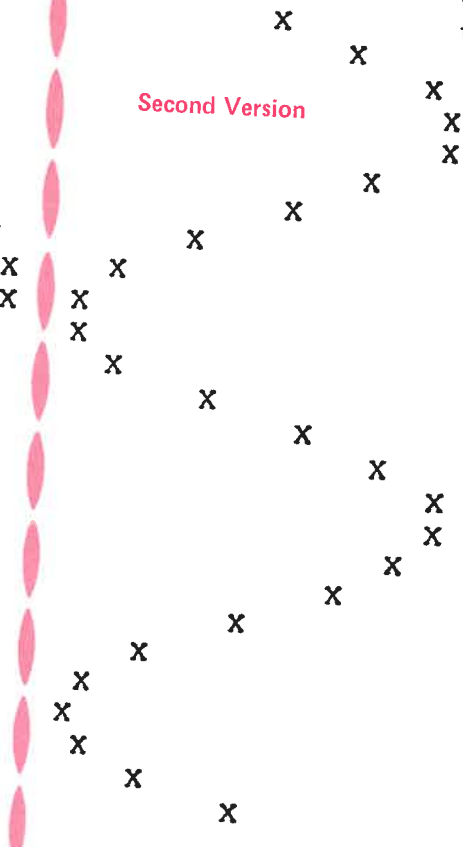
First Version



```
10 FOR P=0 TO 4*3.14159 STEP .5
RUN
```

What did we change to get this output?

Second Version



Modify the program to see
(1) the effect of changing the number of points plotted;
(2) the effect of changing the "scale expansion factor," and
(3) the effect of changing the number of cycles graphed.

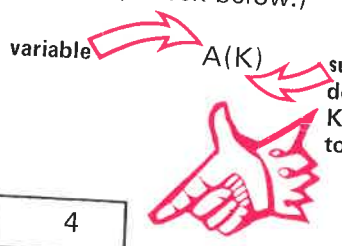
THE REALM OF SUBSCRIPTED VARIABLES

Now we open up for you the last really important capability of BASIC, one that makes complicated, tedious manipulation of information much easier. This is the realm of *subscripted variables.* Subscripted variables are used to keep track of information in lists and arrays. (Not that you have a complete knowledge of BASIC, mind you, but the other nice little things BASIC can do are like tinsle on the tree.)

And Then There's VARIABLE SUBSCRIPTS

READ

One thing that makes these subscripted variables so handy is that they make it easy to assign a lot of values to a lot of variables. And the reason why they make things easy? Because the subscript for a subscripted variable can also be a variable. (Don't panic, check below.)



K	4
A(4)	18

Right off let's be sure we understand that the value of the *subscript* is *not* the value assigned to the *subscripted variable.* This sometimes confuses people at first, so we will make that point several times. In the example above, the value of the *subscript* is 4, and the value of the *subscripted variable* A(4) is 18. Check the boxes.

Don't be afraid to go back and read this introduction over again. When you think you've got the theory down, go on and try out the demonstration programs using subscripted variables.

This program assigns values to four subscripted variables from a DATA statement, then PRINTs the subscripted variables and their assigned values.

```
NEW
```

DO IT

```
5 REM-FIRST SUBSCRIPTED VARIABLE DEMO
10 READ Y(0), Y(1), Y(2), Y(3)
20 ? "Y(0) =";  Y(0)
30 ? "Y(1) =";  Y(1)
40 ? "Y(2) =";  Y(2)
50 ? "Y(3) =";  Y(3)
60 DATA 3, 8, 2, 6.5
RUN
Y(0)  =  3
Y(1)  =  8
Y(2)  =  2
Y(3)  =  6.5
```

If this program won't work for you, and you are absolutely certain that it is typed in correctly, then read on. The original Atari BASIC and some others do not allow you to assign values or strings to subscripted variables in READ or INPUT statements, only in LET direct assignment statements. So you have to use regular variables in the READ or INPUT statement, then follow with LET statements to transfer the assignment to the appropriate subscripted variable. See also page 142. A separate possible problem is that some BASICs do not allow a subscript of zero, as in Y(0). Study page 142 to alert yourself to the simple fixes or program modifications for these BASICs, then return here to make them in this and subsequent examples. And I'd say there's a 50-50 chance that your computer system's BASIC reference materials will help clarify things.

Now Let Me Make This Perfectly CLEAR

Note that the value of A is the value of the FOR-NEXT loop control variable, which increases by 1 (one) each time through the loop. Therefore, the first time through the loop, A = 0 and Y(A) is Y(0). The READ statement then assigns the first value from the DATA statement, 3, to the box labelled Y(0).

Don't confuse the value of the FOR-NEXT loop control variable A with the value of the subscripted variable Y(A), which is assigned from the DATA statement. We're just using the value of control variable A to have the computer assign values to an *array* or *list* of subscripted variables, one after the other. The value of A determines which subscripted variable Y(0) to Y(10) is assigned the next value from the DATA statement. FOR-NEXT loops make it easy to assign a lot of values to a lot of variables, and subscripted variables make it easy to keep track of a lot of values.

A One Liner

Want to try that last little program in multiple statements per line?

Do it like this:

```
10 FOR A=0 TO 10 : READ Y(A) : ? "Y("; A; ") ="; Y(A) : NEXT A
20
30
40
LIST
```

After replacing line 10, just type the line numbers for lines you want to take out, and hit RETURN each time.

Were you brave enough to try this statement without semicolons?

```
5 REM-3RD SUBSCRIPTED VARIABLE DEMO
10 FOR A=0 TO 10 : READ Y(A) : PRINT "Y("; A; ") ="; Y(A) : NEXT A
50 DATA 3, 8, 2, 6.5, 211, 81, 1, -32, 7, .3333, 5

RUN
Y( 0 ) = 3
Y( 1 ) = 8
Y( 2 ) = 2
Y( 3 ) = 6.5
Y( 4 ) = 211
Y( 5 ) = 81
Y( 6 ) = 1
Y( 7 ) =-32
Y( 8 ) = 7
Y( 9 ) = .3333
Y( 10 ) = 5
```

RUN IT — Same Old Results...

(Did you get the same output?)

If you need further convincing, use direct mode to see which string is stored in which subscripted string variable.

DO IT

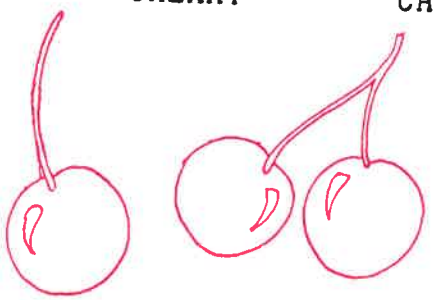
```
? Y$(2), Y$(4), Y$(0), Y$(3)
PEACH          PLUM          APPLE          CHERRY
```

READ

Be clear about this: different variables can be used to determine which subscripted variables is being referred to. It's the *value* of the variable used to indicate the subscript that is important. Do another direct mode statement to illustrate this point. (You haven't typed NEW have you? Well, don't!)

DO IT

```
A=3 : B=3 : C=3 : ? Y$(A), Y$(B), Y$(C)
CHERRY          CHERRY          CHERRY
```



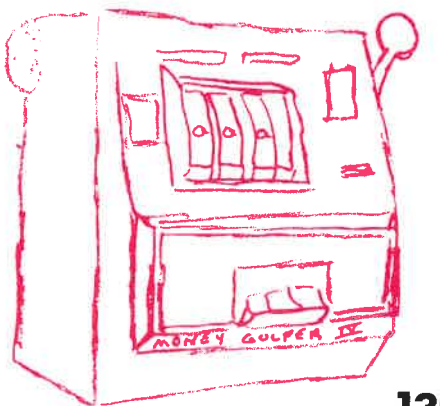
No matter what variable is used to tell the computer the value of the subscript, if the subscripted variable is Y$(3), then the computer will print the string assigned to Y$(3).

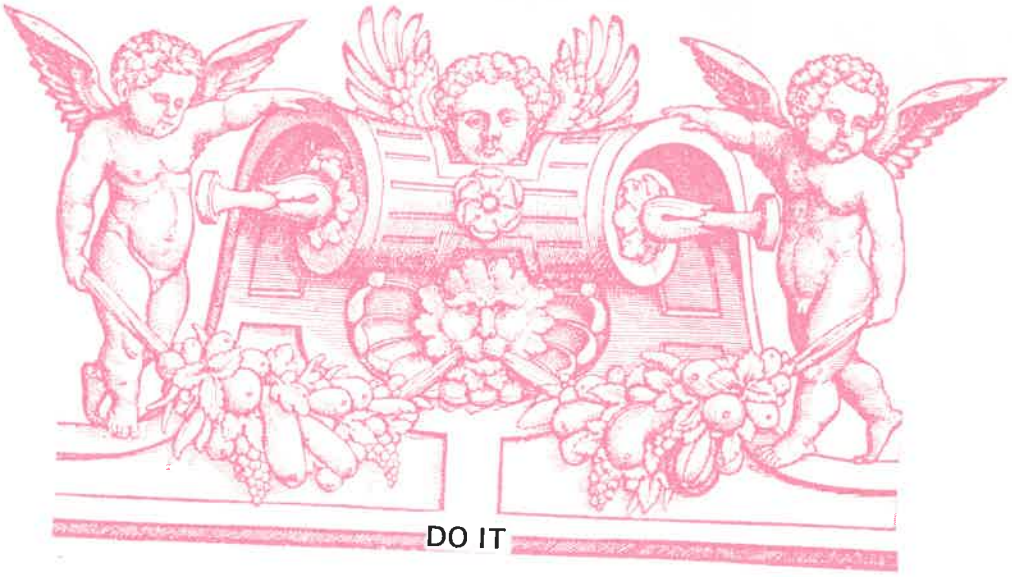
READ

This just gets more and more interesting, and opens more and more programming possibilities. In order to determine the value of a subscript, the computer will even do calculations inside the subscript parentheses. Try it in direct mode, with our fruity list assigned to the Y$ subscripted variables still in the computers memory.

```
X=2 : Y=8 : Z=6 : ? Y$(Y/X), Y$(Z-X), Y$((Y*Z)/(X*Z))
PLUM            PLUM            PLUM
```

Hmmmm, it would seem that every one of those calculations in the subscript parentheses resulted in the same subscript value. Do you agree? Don't you wish this was a slot machine?

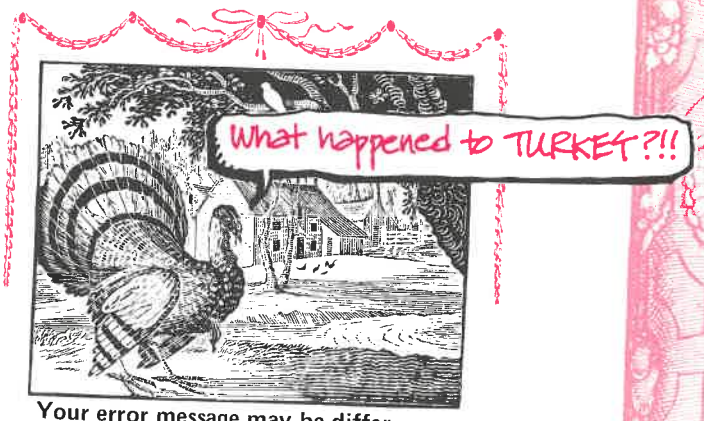




DO IT

Now, a longer list of data in a longer Y$ array or list. Change Lines 10 and 20, and add another DATA statement, like this:

```
10 FOR A=0 TO 11 : READ Y$(A) : ? A; Y$(A) : NEXT A
20
60 DATA BANANA, ORANGE, FIG, APRICOT, TURKEY
LIST

5 REM-SUBSCRIPTED STRING VARIABLE DEMO
10 FOR A=0 TO 11 : READ Y$(A) : PRINT A; Y$(A) : NEXT A
50 DATA APPLE, PEAR, PEACH, CHERRY, PLUM, MELON, GRAPE
60 DATA BANANA, ORANGE, FIG, APRICOT, TURKEY

RUN
 0 APPLE
 1 PEAR
 2 PEACH
 3 CHERRY
 4 PLUM
 5 MELON
 6 GRAPE
 7 BANANA
 8 ORANGE
 9 FIG
 10 APRICOT

?BS ERROR IN 10
```



Your error message may be different. BS means Bad Subscript, and that's no bs. But it is an error message we haven't seen before. For sure it isn't the same as the out of data error message, because there was still another item of DATA left unread when the bad subscript error stopped the RUN. Which is a sneaky way of introducing the next topic . . .

It works just the same for values as for string arrays.

```
NEW


10 FOR B=0 TO 15
20 F(B)=B
30 PRINT "F("; B; ") ="; F(B)
40 NEXT B

RUN
F( 0 ) = 0
F( 1 ) = 1
F( 2 ) = 2
F( 3 ) = 3
F( 4 ) = 4
F( 5 ) = 5
F( 6 ) = 6
F( 7 ) = 7
F( 8 ) = 8
F( 9 ) = 9
F( 10 ) = 10

?BS ERROR IN 20
```

Now add a DIM statement and try the program again.

```
5 DIM F(15)
RUN
F( 0 ) = 0
F( 1 ) = 1
F( 2 ) = 2
F( 3 ) = 3
F( 4 ) = 4
F( 5 ) = 5
F( 6 ) = 6
F( 7 ) = 7
F( 8 ) = 8
F( 9 ) = 9
F( 10 ) = 10
F( 11 ) = 11
F( 12 ) = 12
F( 13 ) = 13
F( 14 ) = 14
F( 15 ) = 15
```

There's always one more thing to learn. Take out Line 20 in the last program, to see what values are stored in the F array if you don't assign any value.

```
20
LIST

5 DIM F(15)
10 FOR B=0 TO 15
30 PRINT "F("; B; ") ="; F(B)
40 NEXT B
```

```
RUN
F( 0 ) = 0
F( 1 ) = 0
F( 2 ) = 0
F( 3 ) = 0
F( 4 ) = 0
F( 5 ) = 0
F( 6 ) = 0
F( 7 ) = 0
F( 8 ) = 0
F( 9 ) = 0
F( 10 ) = 0
F( 11 ) = 0
F( 12 ) = 0
F( 13 ) = 0
F( 14 ) = 0
F( 15 ) = 0
```

BASIC assumes that any variable, including any subscripted variable, has a value of zero until it is told otherwise.

Using an Array to Keep Count

In our simulation program we will use the array T to keep track of how many 1's, 2's, 3's, 4's, 5's and 6's come up on our simulated die.

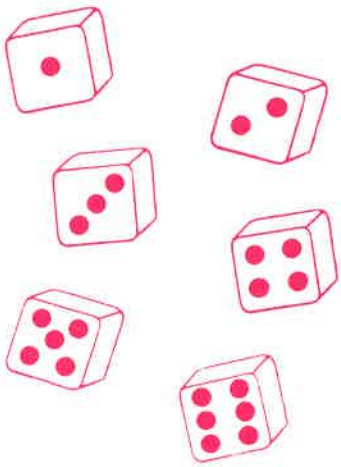
To begin with, all the values of the subscripted variable are 0.

We don't use T(0) since there is no zero on the die.

D	

D will be assigned a value from the RND statement each time it is executed.

T(0)	0
T(1)	0
T(2)	0
T(3)	0
T(4)	0
T(5)	0
T(6)	0



Let's say that the statement (shown on the previous page) that generates the RND integer gives us a 3, so that D = 3.

Each time D = 3 from the RND line, we want the value of T(3) to increase by one to show one roll of the number 3. We can use a "counting" statement with the T array subscripted variable, like this:

T(D) = T(D) + 1 This is the "counting" statement.

If D = 3 then T(3) = T(3) + 1

New value of T(3).

Old value of T(3) plus 1.

D	3

First T(3) = 0.

T(3)	0

Current value of T(3).

Then T(3) = 1

T(3)	1

After T(D) = T(D)+1 is executed for D = 3.

But if D comes up a two (D = 2) then T(2) = T(2) + 1, that is, the value stored at T(2) is increased by 1.

Since there are 6 faces on the die with from 1 to 6 spots, we can use an array or list of 6 to keep track of what face comes up with each simulated throw of the die.

Circle Your Choice

This way of using subscripted variables for counting things can be adapted to many kinds of record keeping. This same way of using subscripts is also a way of tabulating an opinion poll or questionaire, or to count votes, or to score an exam, or ... well, you think of some examples.

Say you want the computer to tabulate or count the responses to this questionaire. It could also be a voting ballot or the answers on a multiple choice test.

Which candidate will you vote for? (Circle the number for your choice.)

1. **Need Some**
2. **Want More**

Here are the votes (or whatever) in DATA statements.

```
900 DATA 1, 1, 1, 2, 1, 2, 1, 2, 2, 1, 2, 2, 2, 2, 1, 2, 1, 2
910 DATA 1, 2, 2, 2, 2, 1, 2, 1, 1, 1, 2, 2, 1, 1, 2, 1, 2, 2, 2
```

Let's have the computer READ them one at a time and have them tallied with our T array (kinda small array ... only two boxes, for a Vote of 1 or a Vote for 2).

```
10 READ V

30 T(V)=T(V)+1
```

T(1)	0
T(2)	0

The values are zero to start with.

V	

Value of V will change for each vote that is read from the DATA statement.

The new value of T(V) = old value of T(V) plus 1 for the vote being tallied. If READ V comes up with V = 1 from the DATA statement, then

$$T(V) = T(V) + 1$$

or $T(1) = T(1) + 1$

The value in box | T(1) | | or | T(2) | | gets kicked up by +1, and to belabor the point, which one gets kicked depends on the value of the subscript V — whether V is 1 or 2 on that trip through the loop. (Pause here for inhalation therapy.)

After each vote is tallied, we want the computer to go back and READ another vote from the DATA statement.

```
40 GOTO 10
```

SALES REPORT

Here is a slightly different way of using a one dimensional array. We call the program Sales Report By Territory. Nationwide Peddlers has six sales territories, with one or more salespersons in each territory. Each salesperson in each territory submits a quarterly report (that's once every 3 months). We want a program to summarize the sales by territory, regardless of how many sales reports are submitted by the sellers in each of the six territories. Each sales report tells (1) which territory the report is from, numbered 1 to 6 and (2) how much was sold (in dollars) by the salesperson. Our DATA statements will hold pairs of DATA, with the first number indicating territory and the second number indicating sales in dollars.

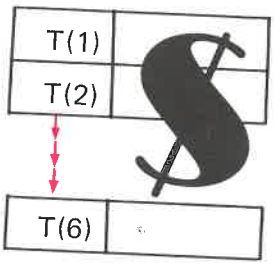
```
100 DATA 1,2350, 4,1750, 2,2000, 1,1345, 5,3200, 3,1220, 6,2100
110 DATA 6,1240, 5,2450, 3,4200, 2,1275, 4,1100, 4,1800 ,3,900
120 DATA 5,2010, 2,1370, 1,1350, 5,1710, 3,2500, -9999,-9999
```

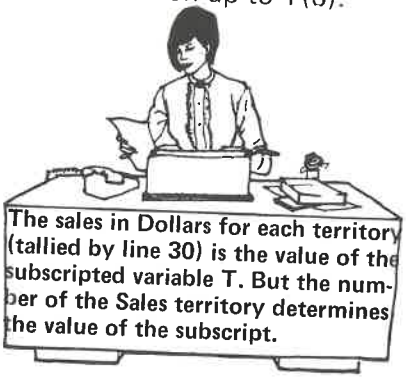
Note the double "end of DATA" flags, because the computer will READ two values at a time, and we wouldn't want an Out of Data error, now would we?

First we DIMension the T array, then READ S, D, where S = Sales territory number, and D is the Dollar sales for one salesperson in that territory. We use the subscript to sort the sales information by territory and keep track of the dollars in sales for each of the six sales territories (see Line 30).

```
5 REM-SALES REPORT BY TERRITORY
10 DIM T(6)
20 READ S,D : IF D=-9999 THEN 40
30 T(S)=T(S)+D : GOTO 20
```

Notice that T(1) keeps track of the sales in dollars for territory 1, T(2) keeps a tally of the sales in dollars for territory 2, and so on up to T(6).





The sales in Dollars for each territory (tallied by line 30) is the value of the subscripted variable T. But the number of the Sales territory determines the value of the subscript.

SALES AND SALARY REPORT

In the next program called Sales and Salary Report, we are using a kind of example, which in larger and more complete form, would be a typical business application of computers, and you know that there is a lot of computing done by businesses of all sizes these days. In the SSR program, we use 3 different subscripted variables, as well as several regular variables and even a string variable. The boxes for the subscripted variables hold information related to eight items that Diligent Industries is peddling through its hot sales force of four salespersons.

Each salesperson is selling the same 8 items. Each month the people on the sales force must each submit a report showing the quantity of each product or item sold. The monthly sales report could be a form like this:

Name _____ Branch Office _____

Item	1	2	3	4	5	6	7	8
Sales								

Salesmen are paid a flat $700 per month plus 10% commission on sales over $3500 each month.

As sales manager, we want our computerized report to show: (1) the total sales in dollars for each salesperson and (2) each salesperson's total salary based on the flat salary plus commission. We also want (3) to show how many of each item or product was sold, and (4) the total dollar income from the sale of each item by all salespersons that month. We also want (5) the grand total of all salaries paid out and all income from sales of DI's great line of products.

You can fix the report display format produced by this program by modifying certain PRINT statement lines to suit your screen display or printer line lengths. Be sure to check lines 130, and 140.

```
NEW

10 DIM P(8), V(8), T(8), Q(8), N$(12)
20 FOR X=1 TO 8 : READ P(X) : NEXT X
30 PRINT "SALESPERSON", "TOTAL SALES", "SALARY"
40 S4=0
50 READ N$ : IF N$="END" THEN 120
60 FOR X=1 TO 8 : READ Q(X)
70 U(X)=U(X)+Q(X)
80 S4=S4 + Q(X)*P(X) : NEXT X
90 PRINT N$, S4,
100 S=S+S4 : IF S4<=3500 THEN PRINT 700 : S1=S1+700 : GOTO 40
110 S3=700+((S4-3500)*.1) : PRINT S3 : S1=S1+S3 : GOTO 40
120 PRINT : PRINT "TOTALS", S, S1 : PRINT
130 PRINT "ITEM", "PRICE/ITEM", "UNITS SOLD", "TOTAL SALES"
140 FOR X=1 TO 8 : PRINT X, P(X), U(X), U(X)*P(X)
150 S2=S2+U(X)*P(X) : NEXT X
160 PRINT : PRINT "GRAND TOTAL OF SALES",, S2
200 DATA 2.05, 18.45, 6.75, 9.95, 25.00, 16.50, 5.50, 12.60
210 DATA D.MILLER, 120, 15, 75, 0, 20, 100, 80, 144
220 DATA B.MIDLER, 160, 1, 90, 55, 16, 120, 96, 132
230 DATA P.PADRE, 80, 10, 60, 40, 5, 75, 10, 55
240 DATA A.XAVIER, 144, 60, 96, 96, 36, 144, 106, 90
250 DATA END

RUN
SALESPERSON     TOTAL SALES     SALARY
D.MILLER         5433.4          893.34
B.MIDLER         6072.4          957.24
P.PADRE          3262            700
A.XAVIER         7998.4          1149.84

TOTALS           22766.2         3700.42

ITEM            PRICE/ITEM      UNITS SOLD      TOTAL SALES
 1               2.05            504             1033.2
 2               18.45           86              1586.7
 3               6.75            321             2166.75
 4               9.95            191             1900.45
 5               25              77              1925
 6               16.5            439             7243.5
 7               5.5             292             1606
 8               12.6            421             5304.6

GRAND TOTAL OF SALES                             22766.2
```

Extra for Experts: modify the program so that decimal points are aligned, and so that one or two zeros are added in the "cents" columns when needed to complete the "dollars and cents" format. If your version of BASIC has the PRINT USING statement, it can help do this job.

6. You have your school transcript in front of you. The college of your future choice says they will accept anyone with 75 A and B grades. Write a program to count your A's and B's. Place all your grades in DATA statements using this scale:

   A = 4 B = 3 C = 2 D = 1 F(n/c) = 5

   Sample DATA statement: DATA 4,3,4,2,2,1,4,3

7. Slot machine simplified: The string array exercise in this chapter with cherries and plums, etc. sets up a natural situation for a slot machine simulation (Nevada style). Write a program that simulates a simple machine and as your energy and interest allows, add to it to make it more sophisticated by fixing the odds, increasing the WIN chances, adding more money to the betting, etc.

   Simple form: From DATA statements read in to an array: CHERRY, BAR, PEACH, PLUM, APPLE. Then choose three of them randomly and print the results.

   If column 1 is a cherry, you win $0.05.
   If column 1 and 2 are the same you win $0.10.
   If column 1, 2 and 3 are the same you win $0.25.
   If columns 1, 2, and 3 are BAR, you win whatever is in the "kitty."

Classic Definition of Classroom Computer Simulation: "Reproducing on a computer an activity or occurrence that would be impossible or impractical to do in a classroom because of cost, time, danger, and so on."

This stacked up arrangement allows us to relate subscripts to particular locations or "boxes" for holding values in rows and columns. Want some vocabulary? The rectangular arrangement of doubly subscripted variables in rows and columns is called a *table* or *matrix*, or *two-dimensional array.* Remember that an array of singly subscripted variables is a *list* or *one-dimensional array.*

This is a list or a one-dimensional array.

B(0)	
B(1)	
B(2)	
B(3)	

One subscript, one dimension.

This is called a table, or two-dimensional array, or matrix:

A(0,0)		A(0,1)		A(0,2)		A(0,3)		A(0,4)		A(0,5)	
A(1,0)		A(1,1)		A(1,2)		A(1,3)		A(1,4)		A(1,5)	
A(2,0)		A(2,1)		A(2,2)		A(2,3)		A(2,4)		A(2,5)	
A(3,0)		A(3,1)		A(3,2)		A(3,3)		A(3,4)		A(3,5)	
A(4,0)		A(4,1)		A(4,2)		A(4,3)		A(4,4)		A(4,5)	

Two subscripts

Two dimensions

Just as for singly subscripted variables, those with two subscripts can have subscripts starting at zero. Up to 30 values or strings may be assigned to this particular array, each identified by its own subscripted variable. Of course, if it were a string array, there would be a $ after the A's like this: A$(3,4).

Arrays created by using variables with double subscripts are handy for storing data in the computer. You *must* tell the computer what the maximum DIMensions of your doubly subscripted variables will be. You may use the same DIM statement to dimension all variables in one statement, whether string variables, singly or doubly subscripted variables:

```
20 DIM C(3,4), X$(15), Y(12), M$(F,S)
```

line number

2 dim array

1 dim string array

1 dim array

2 dim string array

The exceptions (what, again?)—if things aren't working right when you use DIM, refer to pages 25, 27, 142, 143, and the reference manual for your BASIC.

Now fill in the values assigned to these subscripted variables by the last program. We did the first two.

C(0,0)	5	C(0,1)	10	C(0,2)		C(0,3)	
C(1,0)		C(1,1)		C(1,2)		C(1,3)	
C(2,0)		C(2,1)		C(2,2)		C(2,3)	

Now let's start automating. As we did with one-dimensional arrays before, we use FOR-NEXT loops to give values to the subscripts, and thus tell the computer which subscripted variable it is to deal with. But now we have 2 subscripts, so we use *nested* FOR-NEXT loops to go through the possible values.

DO IT

Replace Lines 50, 60, and 70 in the last program, and add Line 80.

```
50 FOR R=0 TO 2
60 FOR C=0 TO 3
70 ? "C("; R; ","; C; ") ="; C(R,C); "    ";
80 NEXT C : ? : NEXT R
LIST

5 REM-DOUBLE SUBSCRIPT DEMO
10 DIM C(2,3)
20 READ C(0,0), C(0,1), C(0,2), C(0,3)
30 READ C(1,0), C(1,1), C(1,2), C(1,3)
40 READ C(2,0), C(2,1), C(2,2), C(2,3)
50 FOR R=0 TO 2
60 FOR C=0 TO 3
70 PRINT "C("; R; ","; C; ")="; C(R,C); "  ";
80 NEXT C : PRINT : NEXT R
100 DATA 5, 10, 88, -19, 100, 8.25, 91, 22, -1.5, 15, 9, 2
```

READ

```
80 NEXT C : ? : NEXT R
```

Used to "cancel" the ; at the end of Line 70 and start a new line of output.

Did you have trouble entering Line 70 correctly? Let's look at it more closely.

```
70 ? "C("; R; ","; C; ")="; C(R,C); "  ";
```

- `?` — The first part of the subscripted variable notation.
- `R` — Value of first subscript.
- `","` — PRINT the comma used to separate subscripts.
- `C` — Value of second subscript.
- `")="` — The subscript parentheses closed.
- `C(R,C)` — PRINT the value assigned to this subscript variable.
- `"  "` — Leave spaces between the subscripted variables and values.
- `;` — Stay on the same line.

Lots of extra semicolons here that could be omitted in some versions of BASIC.

In some of the spiffier BASICs, you don't have to put in all those semicolons

SUPERTALLY

Let's go back to our vote counting program. Using variables with double subscripts allows us to add another dimension to tally or count up for us. Our new questionaire asks "Who did you vote for in the last election?"

Question 1: Who did you vote for in the last election? Circle the number by your choice.

1. Powerman
2. Moneyman
3. Other

Question 2: Circle the number by your age group.

1. 18 – 29
2. 30 – 39
3. 40 – 49
4. 50 or over

We want to write a program to summarize the data gathered from this poll. Note that there are two questions, and 3 possible answers for the first, and 4 possible answers for the second. Any one questionaire yeilds two answers, the answer to Question 1 (which can be a 1, 2 or 3), and the answer to Question 2 (which can be a 1, 2, 3, or 4.) Obviously we want to know the answers as related to age groups, as well as the totals of votes in each catagory from Question 1 for all age groups, just like the big time pollsters. (Gallup Poll has nothing on us!)

Remember how we used the subscript of a variable to decide which catagory to tally a vote in?

```
10 READ V
30 T(V)=T(V)+1
```

T(1)	
T(2)	

Now we have two answers to tally, so we use a two dimensional array.

	18 – 29	30 – 39	40 – 49	50 or over
Powerman	C(1,1)	C(1,2)	C(1,3)	C(1,4)
Moneyman	C(2,1)	C(2,2)	C(2,3)	C(2,4)
Other	C(3,1)	C(3,2)	C(3,3)	C(3,4)

So far, this much of the program (plus the DATA statements) will count all the responses to the questionaire.

```
LIST

5 REM-VOTE COUNTING WITH TWO-DIMENSIONAL ARRAY
10 DIM C(3,4)
20 READ V,A : IF A=-9999 THEN 40
30 C(V,A)=C(V,A)+1 : GOTO 20
900 DATA 1,2, 1,3, 2,2, 2,3, 2,1, 3,2, 3,4, 2,3, 2,3, 3,2, 1,4
910 DATA 3,1, 3,2, 1,4, 2,4, 2,3, 1,3, 2,4, 1,4, 2,2
920 DATA 2,1, 2,2, 3,2, 2,4, 1,2, 1,3, 2,4, 1,4, 2,3, 3,1, 3,3
930 DATA -9999, -9999
```

The next section of the program must tell the computer to tell us what it has counted. We want the following information when the program is RUN.

```
RUN
CANDIDATE      18-29     30-39     40-49     50 +

POWERMAN         0         2         3        4
MONEYMAN         2         3         5        4
OTHER            2         4         1        1
```

First a statement to print the headings and a blank line. Substitute semicolons and TABs for the commas, if you are working with a 40 column display or printer.

```
40 ? "CANDIDATE", "18-29", "30-39", "40-49", "50 +" : ?
```

And now the information stored in the C array. If you guessed that we will use nested FOR-NEXT loops, you are as brilliant as you look. But what about the candidates' names? Let's put those in a DATA statement, and assign them to a string variable, C$. If we place the READ C$: PRINT C$ instructions *inside* the first loop, but *before* the second loop, they will be printed at the beginning of each line in the table. See page 106.

```
50 FOR V=1 TO 3 : READ C$ : ? C$,
60 FOR A=1 TO 4 : ? C(V,A), : NEXT A,V
940 DATA POWERMAN, MONEYMAN, OTHER
```

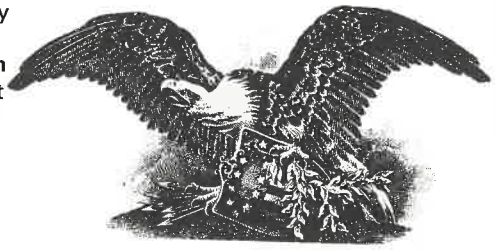
You can automate the TAB settings inside the loops (for printing the C array values under the proper headings within a 40 column format) with this technique:

```
50 FOR V = 1 TO 3 : READ C$ : ? C$; : LET M = 8
60 FOR A = 1 TO 4 : ? TAB(M); C(V,A); : LET M = M + 8 :
   NEXT A : NEXT V
```

There is one more peice of information that we might want to have reported: the total number of people answering the questionaire. Let's just add on one more little program segment. Again we take advantage of the fact that we haven't used all the locations of the C(V,A) array with zero as subscript. To tally the C(V,0) values and store the total number of respondents to our poll, in C(0,0), we use anosther FOR-NEXT loop.

```
110 ? "TOTAL POLLED:";
120 FOR V=1 TO 3 : C(0,0)=C(0,0)+C(V,0) : NEXT V : ? C(0,0)
RUN
CANDIDATE       18-29          30-39          40-49          50 +

POWERMAN          0              2              3              4
MONEYMAN          2              3              5              4
OTHER             2              4              1              1

TOTALS:
ANSWER 1 : 9
ANSWER 2 : 14
ANSWER 3 : 8
TOTAL POLLED: 31
```

You can fix the report display format produced by this program by modifying certain PRINT statement lines to suit your screen display or printer line lengths.



READ 

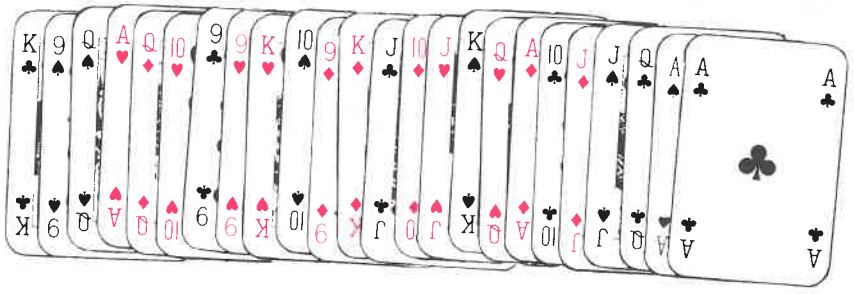
Of course there are other kinds of information that we might get from this data and our program, with further additions. You might write a routine (program segment) to tally how many people in each age group was polled. If you know statistics, you might try writing some routines to provide a statistical analysis of the program, or percentages in the various catagories. Use your imagination plus the knowledge of BASIC that you have worked on hard to learn.

OTHER DIMENSIONS

Other Dimensions

You don't have to stop with 2 dimensional arrays. In fact, many Microsoft BASICs allow up to 255 dimensions.

However, because of the limitations on the size of a statement line, you can't always get *that* many dimensions into a statement.



D(4,13)...the subscripted variable to keep track of which cards have been dealt by checking and "marking" in an array, so that the computer will not deal the same card twice. There are 13 cards in 4 suits — note the subscripts in the array D(4,13). The value of the subscripts are determined by two RND numbers, and that is how the computer will know which card is being dealt.

J.....the suit of the card to be dealt (See Line 110).

K.....the number (1 to 13) of the card to be dealt. See line 100.

```
99 REM-CARD DEALING SUBROUTINE, LINES 100-330
100 K=INT(13*RND(1))+1
110 J=INT(4*RND(1))+1
120 IF D(J,K)=-1 THEN 100
130 D(J,K)=-1
```

In Line 130, the values of J and K determine which D(J,K) box will be marked or set to −1. We could have used any value other than zero to indicate that a particular box has been changed from zero. But before the box is marked −1, the computer checks to see if that "card" has already been dealt, that is, if the box for D(J,K) was already marked with a −1. Line 120 sends the computer back to try again with another set of RND numbers if there was already a −1 marked in the box.

In this program, each card has a number for its face value, "selected" at random from 1 to 13 by Line 100.

```
100 K=INT(13*RND(1))+1
```

Number:	1	2	3	4	5	6	7	8	9	10	11	12	13
Card:	Ace	2	3	4	5	6	7	8	9	10	JACK	QUEEN	KING

Each card has a number (1 to 4) for its suit, selected at random by Line 110.

```
110 J=INT(4*RND(1))+1
```

Number:	1	2	3	4
Suit:	clubs	spades	hearts	diamonds

Line 120 checks to make sure that card D(J,K) hasn't been dealt, and if it hasn't, Line 130 notes with a −1 that it is about to deal card D(J,K).

ON...GOTO...

Lines 140 and 150 introduce the ON ... GOTO statement.

ON _____ GO TO ___, ___, ___, ___, ___, ___

A variable or expression to calculate.

All these must be line numbers that are in the program, and they must be separated by commas.

As many line numbers as can fit in a statement line, can follow an ON ...GOTO statement. If the ON ... GOTO variable is 1, then the computer *GOes TO* the first line number. If the variable value is 2, then it goes to the second line number listed, and so on. However, if the value of the ON ... GOTO variable is negative, or is zero, *OR* is bigger than the number of line numbers after GOTO, then the computer just skips on ("falls through the statement," they say) to the next line numbered statement in the program in some BASICs.

This is handy, because you may wish to use more line numbers then will fit in one line. The trick is this: Say that only the first 10 line numbers in Line 140 actually fit on the line —

```
140 ON K GOTO 200,210,210,210,210,210,210,210,210,210
```

We leftoff the last 3 lines numbers, pretending they wouldn't fit. We tell our next ON ... GOTO variable to pick up where the last one left off, like this:

```
145 ON K-10 GOTO 220,230,240
```

Subtract the largest possible value of K for the last line. K would have to be greater than 10 in order to "fall through" the last statement and arrive at this line.

Like GOSUB and RETURN, ON ... GOTO must be the last statement if used in a multiple statement line, or else it must stand alone.

In Line 140 we find

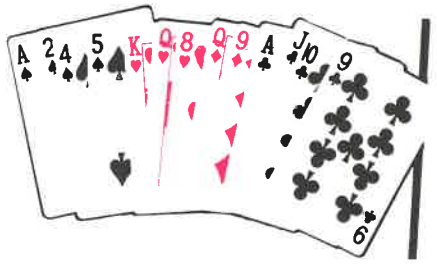
```
140 ON K GOTO 200,210,210,210,210,210,210,210,210,210,220,230,240
```

This line selects which statement will be used to PRINT the card selected. If K = 1, then the computer goes to Line 200 and prints ACE. If K = 2, 3, 4, 5, 6, 7, 8, 9, or 10, then the computer is sent to Line 210 and prints the value of K (the number of the card being dealt). If K= 11, 12, or 13, the computer goes to Lines 220, 230 or 240 and prints JACK, QUEEN, or KING.

Notice that ON ... GOTO can sometimes substitute for bunches of IF...THEN statements.

The ON ... GOTO statement in Line 150 selects the name of the suit to be printed. We sure got double duty out of those RND value for J and K, didn't we?

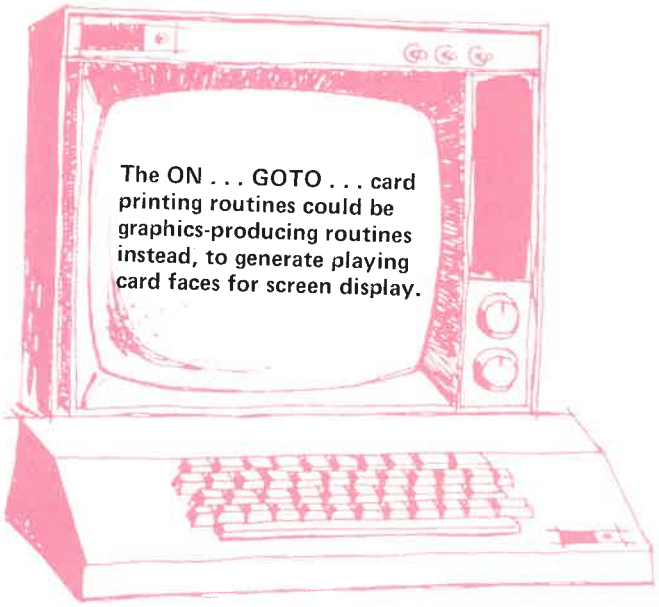
```
LIST

 5 REM-POKER DEALER
 10 DIM D(4,13)
 20 FOR H=1 TO 5 : GOSUB 100
 30 NEXT H
 40 PRINT : INPUT "ANOTHER HAND, SAME DECK"; A$ : IF A$="YES" THEN 20
 50 PRINT : INPUT "ANOTHER HAND, NEW DECK"; A$ : IF A$="NO" THEN END
 60 FOR J=1 TO 4 : FOR K=1 TO 13 : D(J,K)=0 : NEXT K,J : GOTO 20
 99 REM-CARD DEALING SUBROUTINE, LINES 100-330
 100 K=INT(13*RND(1))+1
 110 J=INT(4*RND(1))+1
 120 IF D(J,K)=-1 THEN 100
 130 D(J,K)=-1
 140 ON K GOTO 200,210,210,210,210,210,210,210,210,210,220,230,240
 150 ON J GOTO 300,310,320,330
 200 PRINT " ACE "; : GOTO 150
 210 PRINT K; : GOTO 150
 220 PRINT " JACK "; : GOTO 150
 230 PRINT " QUEEN "; : GOTO 150
 240 PRINT " KING "; : GOTO 150
 300 PRINT "OF CLUBS" : RETURN
 310 PRINT "OF SPADES" : RETURN
 320 PRINT "OF HEARTS" : RETURN
 330 PRINT "OF DIAMONDS" : RETURN
```



```
RUN
 QUEEN OF HEARTS
 8 OF HEARTS
 KING OF DIAMONDS
 10 OF HEARTS
 6 OF SPADES
```

EXTRA FOR EXPERTS: Expand the program so that the computer keeps count of the number of cards dealt, and automatically "shuffles the deck" (sets the array to all zeros) after 52 cards are dealt and before the computer tries to deal the 53rd card. In this program, the computer will loop from 120 to 100 forever looking for a place without a −1 in the array, after dealing 52 cards. How about a program to deal blackjack, or score poker hands?

```
ANOTHER HAND, SAME DECK? YES
 3 OF HEARTS
 4 OF CLUBS
 JACK OF SPADES
 9 OF HEARTS
 2 OF DIAMONDS

ANOTHER HAND, SAME DECK? NO

ANOTHER HAND, NEW DECK? YES
 JACK OF DIAMONDS
 8 OF SPADES
 JACK OF CLUBS
 7 OF CLUBS
 QUEEN OF DIAMONDS

ANOTHER HAND, SAME DECK? YES
 QUEEN OF HEARTS
 6 OF SPADES
 3 OF CLUBS
 4 OF SPADES
 7 OF DIAMONDS

ANOTHER HAND, SAME DECK? NO

ANOTHER HAND, NEW DECK? NO
```



FUNCTIONS

RND(X) – generates a random number between 0 and 1

For RND in Microsoft-style BASICs, see pages 110-119.

Following functions apply to most versions of BASIC.

LEN(B$) – gives an integer equal to the number of characters in the string variable.

DEF FNA(A) – define your own function.

ABS(X) – gives absolute value of expression X.

INT(X) – gives largest integer less than or equal to argument X.

TAB(X) – spaces to the specified print column.

USR(X) – calls machine language subroutine X.

FRE(0) – gives the number of bytes unused in memory.

SPC(X) – prints X number of blank spaces.

SGN(X) – gives 1 if X is greater than 0, zero if X is zero, and –1 if X is less than zero.

SIN(X) – gives sine of expression X if X is in radians.

SQR(X) – gives square root of X.

TAB(X) – spaces to the specified print column on terminal.

ATN(X) – gives arctangent of X in radians.

COS(X) – gives cosine of X in radians.

LOG(X) – gives natural (base e) log of X.

POS(X) – gives current position of terminal printhead or cursor.

DEF FN

Allows user to define functions.

line no. DEF FN (variable name)(dummy variable) = function

Some versions of BASIC have:

FIX(X) – returns the truncated value of X.

LOG10(X) – gives common log of X.

PI – constant value of pi, 3.1415927 (π)

RND – generates a random number between 0 and 1. Same sequence each RUN. Use RANDOMIZE statement (10 RANDOMIZE) or RANDOM (10 RANDOM) to change sequence.

STRING FUNCTIONS

INSTR(N1,A$,B$) searches string A$ for substring B$ beginning at character N1 in string A$. Gives zero if substring not found. Gives character position if substring is found.

SPACE$(N) – inserts N spaces within a string.

For Microsoft BASIC string functions, see pages 87-91.

Dear Reader,

Well, you have arrived at the end of this book, but you haven't arrived at the end of BASIC. Your computer system undoubtedly has a version of BASIC with more capabilities than we have covered here. However, if you understand and can use the BASIC instructions and functions we have presented, you are well on your way. If you have done the end of chapter problems (you didn't skip them, did you?), then you are really on your way to being able to write programs to meet your own needs.

The author would appreciate your comments, suggestions, and criticisms. Did you find mistakes? What confused you? Are there better ways to explain things? Did your version of BASIC throw you any curves that we didn't forewarn you of? The author and publishers will continue to revise and improve this book, so your comments really will be read, and the author really does answer letters when they reach him in a timely fashion. When writing, please give a complete description of your computer system and the version of BASIC it uses.

So where to from here? Start with a thorough review of all your computer system's reference materials. If you have a disk drive or cassette that can store data files, then your choice for an intermediate level self-instructional guide could be Data File Programming in BASIC *(Microsoft BASIC-80, Models I and III Radio Shack BASIC, and with North Star BASIC annotations), or* Apple BASIC: Data File Programming, *or* TRS-80 Data File Programming, *all by LeRoy Finkel and Jerald R. Brown.*

Thanks! Have fun and keep on hackin'.

Jerald R. Brown
dilithium Press
11000 S.W. 11th Street, Suite E
Beaverton, Oregon 97005
Toll free phone: 1-800-547-1842

Rainbow's End Farm
Sebastopol, California
November, 1981

Good luck!

The Bobcat

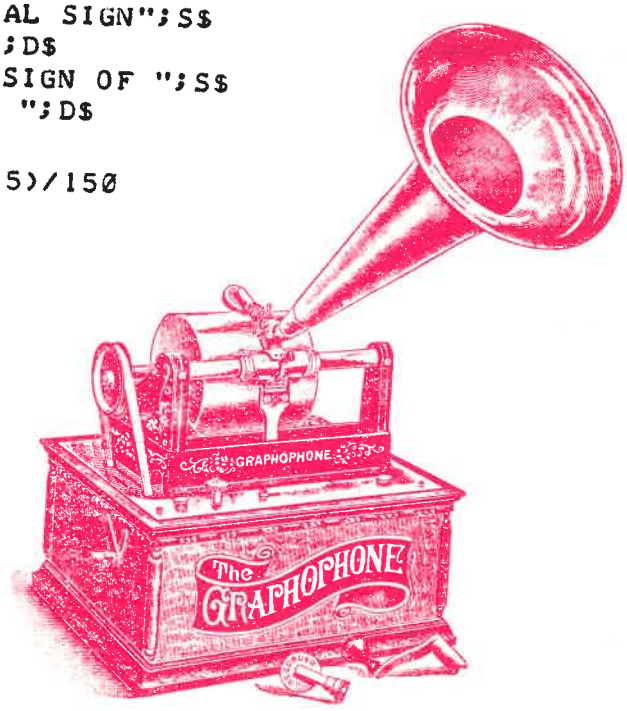
The author holds a B.S. degree in psychology, and an M.Ed. in Research in Instruction from Harvard. He is a co-founder of People's Computer Company. In addition to being the co-author of a dozen instructional books in computer programming, he is also a filmmaker, a gardener, and has done a variety of educational television productions.

5.
```
10 READ A$,B$,A
20 PRINT A$,B$,A
30 DATA "SHERRY DELIGHT","800-555-1212",23

RUN
SHERRY DELIGHT                 800-555-1212    23
```

6.
```
10 INPUT"ENTER YOUR NAME";A$
20 INPUT "ENTER YOUR ASTROLOGICAL SIGN";S$
30 INPUT "ENTER YOUR BIRTHDATE";D$
40 PRINT A$,"YOU ARE UNDER THE SIGN OF ";S$
50 PRINT"SINCE YOUR BIRTHDAY IS ";D$
```

7. (a)
```
10 LET X=(((4.80/.48)*100)*7.5)/150
20 PRINT X,"MONTHS"
RUN
 50                MONTHS
```

(b) No



CHAPTER 3 SOLUTIONS

1.
896700
1,000,000,000,000,000
.001
61,576,200,000
.00000387124

2.
1.78643235E+09
3.13457532E+09
1.2479E-04
7.77777778E-03

3. Hold the CONTROL (or CTRL) key down and type C, then release both keys and hit RETURN (or its equivalent on your keyboard).

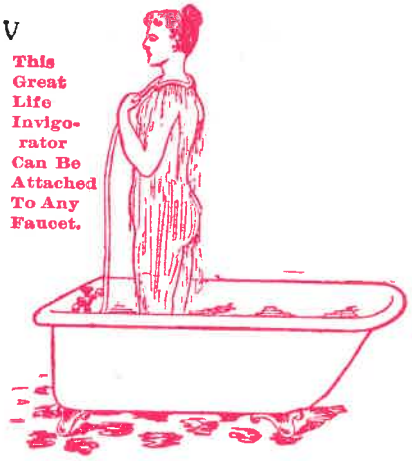
4. CONTROL/C

5.
```
10 PRINT "NUMBER","SQUARED"
20 LET T=1
30 PRINT T,T↑2
40 LET T=T+1
50 GOTO 30
```

```
NUMBER          SQUARED
 1               1
 2               4
 3               9
 4               16
 5               25
 6               36
 7               49
 8               64
 9               81
 10              100
```

6.
```
10 PRINT "PRESS RETURN TO CONTINUE"
20 PRINT "NUMBER","SQUARED"
30 LET T = T + 1
40 PRINT T, T^2
50 INPUT ""; R$
60 GOTO 30
RUN
```

5.
```
20 INPUT"ENTER NO. OF SHOWERS PER DAY";S
30 INPUT "ENTER NO. OF MINS. PER SHOWER";M
40 INPUT"ENTER NO. OF TUB BATHS PER DAY";B
50 INPUT"ENTER NO. OF HAND DISHWASHING JOBS/DAY";D
60 INPUT"ENTER NO. OF AUTO. DISHWASHING JOBS/DAY";A
70 INPUT"NO. OF TOILET FLUSHES/DAY";F
80 INPUT"NO. OF WASHER LOADS/WEEK";W
90 INPUT"NO. OF OUTDOOR HOSE MINUTES/DAY";H
100 INPUT"ENTER NO. OF VARIOUS GALLONS USED/DAY";V
110 LET T=0
120 READ X
130 LET T=T+(S*(M*X))*30
140 READ X
150 LET T=T+(B*X)*30
160 READ X
170 LET T=T+(D*X)*30
180 READ X
190 LET T=T+(A*X)*30
200 READ X
210 LET T=T+(F*X)*30
220 READ X
230 LET T=T+(W*X)*4.2
240 READ X
250 LET T=T+(H*X)*30
260 LET T=T+(V*30)
270 PRINT"YOU USE APPROX. "; T;"GALLONS / MONTH OR ";T/7.5; "CUBIC FEET
280 PRINT"THAT IS AN AVERAGE OF ";T/30;"GALLONS PER DAY"
300 DATA 6,20,15,16,6,35,10
```



Cline's Portable Shower Bath.

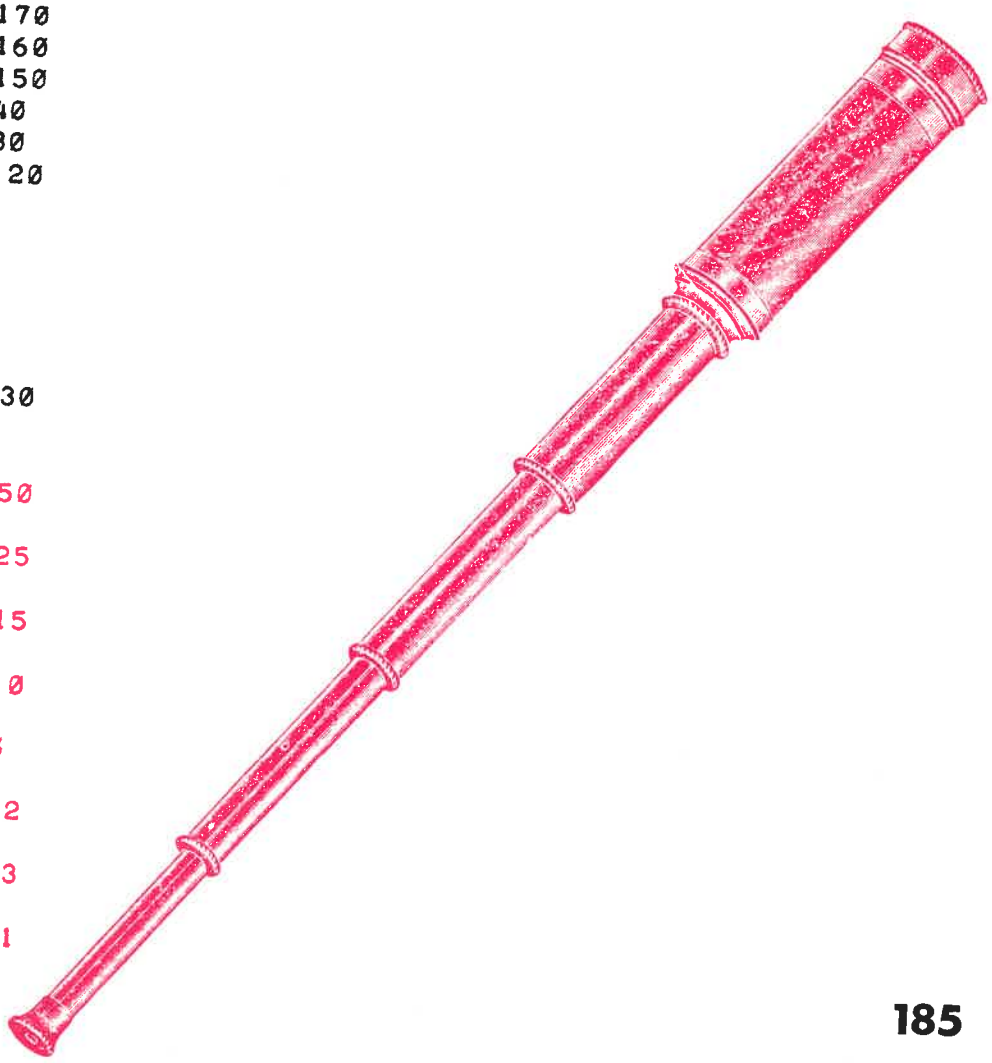
```
RUN
ENTER NO. OF SHOWERS PER DAY? 1
ENTER NO. OF MINS. PER SHOWER? 5
ENTER NO. OF TUB BATHS PER DAY? 1
ENTER NO. OF HAND DISHWASHING JOBS/DAY? 1
ENTER NO. OF AUTO. DISHWASHING JOBS/DAY? 1
NO. OF TOILET FLUSHES/DAY? 5
NO. OF WASHER LOADS/WEEK? 3
NO. OF OUTDOOR HOSE MINUTES/DAY? 0
ENTER NO. OF VARIOUS GALLONS USED/DAY? 50
YOU USE APPROX.  5271 GALLONS / MONTH OR  702.8 CUBIC FEET
THAT IS AN AVERAGE OF  175.7 GALLONS PER DAY
```

6.
```
10 INPUT "ENTER WATER USED IN GALLONS";G
20 LET T=(G/7.5)/100:LET T=(T*.50)+2.85:PRINT"BILL IS "T:GOTO 10

RUN
ENTER WATER USED IN GALLONS? 6000
BILL IS  6.85
ENTER WATER USED IN GALLONS? 10000
BILL IS  9.51667
ENTER WATER USED IN GALLONS? 20000
BILL IS  16.1833
ENTER WATER USED IN GALLONS?
```

5.
```
10 REM CRAPS
20 LET A=INT(6*RND(1)+1):LET B=INT(6*RND(1)+1)
25 PRINT"POINT IS "A+B
30 IF A+B=7 THEN PRINT"WINNER":GOTO 20
40 IF A+B=11 THEN PRINT"WINNER":GOTO 20
50 LET C=INT(6*RND(1)+1):LET D=INT(6*RND(1)+1)
55 PRINT C+D,
60 IF A+B=C+DTHEN PRINT"WINNER":GOTO 20
70 IF C+D=7 THEN PRINT"YOU CRAPPED OUT":GOTO 20
80 GOTO 50

RUN
POINT IS  6
 4                5                10               4                8
 8                9                9                6                WINNER
POINT IS  11
WINNER
POINT IS  6
 7                YOU CRAPPED OUT
POINT IS  5
```

Use the Press RETURN To Continue technique for video displays to slow down the dice-shooting.

6.
```
10 REM STARS
20 LET N=INT(100*RND(1)+1)
30 INPUT"ENTER YOUR GUESS";G:IF G=N THEN PRINT"WINNER":GOTO 20
40 LET D=ABS(G-N)
50 IF D>=64 THEN 170
60 IF D>=32 THEN 160
70 IF D>=16 THEN 150
80 IF D>=8 THEN 140
90 IF D>=4 THEN 130
100 IF D>=2 THEN 120
110 PRINT"*";
120 PRINT"*";
130 PRINT"*";
140 PRINT"*";
150 PRINT"*";
160 PRINT"*";
170 PRINT"*":GOTO 30

RUN
ENTER YOUR GUESS? 50
**
ENTER YOUR GUESS? 25
****
ENTER YOUR GUESS? 15
*****
ENTER YOUR GUESS? 10
*******
ENTER YOUR GUESS? 8
******
ENTER YOUR GUESS? 12
*******
ENTER YOUR GUESS? 13
******
ENTER YOUR GUESS? 11
WINNER
ENTER YOUR GUESS
```



```
OK, I HAVE A NUMBER. START GUESSING.

WHAT IS YOUR GUESS?50
TOO BIG. GUESS AGAIN.

WHAT IS YOUR GUESS?25
TOO BIG. GUESS AGAIN.

WHAT IS YOUR GUESS?20
TOO BIG. GUESS AGAIN.

WHAT IS YOUR GUESS?5
TOO SMALL. GUESS AGAIN.

WHAT IS YOUR GUESS?17
TOO BIG. GUESS AGAIN.

WHAT IS YOUR GUESS?12
TOO BIG. GUESS AGAIN.

WHAT IS YOUR GUESS?11
TOO BIG. GUESS AGAIN.

WHAT IS YOUR GUESS?10
TOO BIG. SORRY, THAT'S 8 GUESSES. LET'S PLAY AGAIN.

OK, I HAVE A NUMBER. START GUESSING.

WHAT IS YOUR GUESS?
```

4. 10 FOR X = 1 TO 72 : PRINT "*" ; NEXT

Note this: The second FOR variable parameter (upper limit) should be exactly equal to the number of characters for a print or display line. See your page 11 test, and substitute this value in the program.

5. 80 column printer or display version.

```
5 REM INTEREST TABLE
8 PRINT"YEARS","5%","5.5%","6%","6.5%"
10 FOR Y=5 TO 25 STEP 5
15 PRINT Y,
20 FOR I=5 TO 6.5 STEP .5
30 PRINT 10000*(1+(I/100))↑Y,:NEXT I:PRINT
40 NEXT Y
```

YEARS	5%	5.5%	6%	6.5%
5	12762.8	13069.6	13382.3	13700.9
10	16289	17081.4	17908.5	18771.4
15	20789.3	22324.7	23965.6	25718.4
20	26533	29177.5	32071.3	35236.5
25	33863.6	38133.7	42918.7	48277.1

6. C = consonant, V = vowel.

```
5 REM RANDOM NAME GENERATOR(CVCCVC)
10 V$="AEIOU":N$="BCDFGHJKLMNPQRSTVWXYZ"
15 FOR N=1 TO 20
20 V1=INT(5*RND(1))+1:V2=INT(5*RND(1))+1
30 C2=INT(21*RND(1))+1:C3=INT(21*RND(1))+1:C4=INT(21*RND(1))+1
40 PRINT"J";MID$(V$,V1,1);MID$(N$,C2,1);MID$(N$,C3,1);
45 PRINT MID$(V$,V2,1);MID$(N$,C4,1),
50 NEXT N
```

```
RUN
JIPFID          JUDBUV          JIDRID          JUHWEK          JEPTIZ
JOJWEB          JAXJOK          JEPSIG          JEZSUK          JIGNAV
JUSPOH          JEPXAT          JIYBOT          JUHLIF          JEMJEK
```

NEXT

CHAPTER 9 SOLUTIONS

1. (a) Line 70
   (b) Print salesman name and total sales
   (c) Tallies up grand total sales
   (d) Line 100 or 110

2. L. FRENCH
   B. MIDLER
   14 00
   500

3. 16
   9
   9
   5

4. Only when your array exceeds 11 variables or elements, in Microsoft-style BASICs; for *all* arrays in some other BASICs.

5. If your BASIC doesn't include leading and trailing spaces for PRINTed values, include spaces inside quotation marks (" ") in line 70, to separate the candidate's "code" numbers from the candidate vote count under the heading for each office.

```
20 READ P1,V1,T1:IF P1=-1 THEN 50
30 P(P1)=P(P1)+1:V(V1)=V(V1)+1:T(T1)=T(T1)+1:GOTO 20
50 PRINT"PRESIDENT","VEEP","TREASURER"
60 FOR X=1 TO 3
70 PRINTX;P(X),X;V(X),X;T(X)
80 NEXT X
90 DATA 1,1,1,2,2,2,3,3,3,1,2,3,3,2,1,2,1,3,2,2,3,1,2,1,-1,-1,-1
```

```
RUN
PRESIDENT       VEEP            TREASURER
 1   3           1   2           1   3
 2   3           2   5           2   1
 3   2           3   1           3   4
```

CHAPTER 10 SOLUTIONS

1. Always

2. S stands for Sex (1 for male, 2 for female)
   A stands for Age group (1 to 6)

80 column printer or display version.

```
10 REM EQUAL OPPORTUNITY
15 DIM C(6,2)
20 READ S,A:IF S=-1 THEN 40
30 C(A,S)=C(A,S)+1:C(0,S)=C(0,S)+1:C(A,0)=C(A,0)+1:GOTO 20
40 PRINT"AGE","MALE","FEMALE","TOTAL"
50 FOR A=1 TO 6:READ D$:PRINTD$,
60 FOR S=1 TO 2:PRINT C(A,S),:NEXT S
70 PRINTC(A,0):NEXT A
80 PRINT"TOTALS",C(0,1),C(0,2)
90 DATA 1,3,2,2,1,1,2,4,2,5,2,6,1,6,1,5,1,4,1,3,1,2,1,2,-1,-1
100 DATA UNDER 21,21-29,30-39,40-49,50-59,OVER 59
```

```
RUN
AGE             MALE          FEMALE        TOTAL
UNDER 21         1             0             1
21-29            2             1             3
30-39            2             0             2
40-49            1             1             2
50-59            1             1             2
OVER 59          1             1             2
TOTALS           8             4
```

40 column display version.

```
10  REM   EQUAL OPPORTUNITY
20  DIM C(6,2)
30  READ S,A: IF S =  - 1 THEN 50
40 C(A,S) = C(A,S) + 1:C(0,S) = C(0,S) + 1:C(A,0) = C(A,0) + 1: GOTO 30
50  PRINT "AGE"; TAB( 10);"MALE"; TAB( 18);"FEMALE"; TAB( 26);
    "TOTAL": PRINT
60  FOR A = 1 TO 6: READ D$: PRINT D$;:T = 12
70  FOR S = 1 TO 2: PRINT  TAB( T);C(A,S);: LET T = T + 8: NEXT S
80  PRINT  TAB( T);C(A,0): PRINT : NEXT A
90  PRINT "TOTALS"; TAB( 12);C(0,1); TAB( 20);C(0,2)
100  DATA 1,3,2,2,1,1,2,4,2,5,2,6,1,6,1,5,1,4,1,3,1,2,1,2,-1,-1
110  DATA UNDER 21,21-29,30-39,40-49,50-59,OVER 59
```

4. In version one, you may need to insert the following statement:
75 PRINT

80 column printer or display version.

```
10 REM SHOPPING GUIDE
15 DIM P(10,4),T(4)
18 REM READ IN DATA
20 FOR S=1 TO 4
30 FOR I=1 TO 10: READ P(I,S):NEXT I:NEXT S
40 REM PRINT IT OUT
50 PRINT"ITEM #","STORE1",2,3,4
60 FOR I=1 TO 10:PRINTI,
70 FOR S=1 TO 4:PRINTP(I,S),:T(S)=T(S)+P(I,S):NEXT S
80 NEXT I
90 PRINT"TOTALS",:FOR X=1 TO 4:PRINTT(X),:NEXT X
100 DATA 1,2,3,4,5,6,7,8,9,10
110 DATA 10,9,8,7,6,5,4,4,2,1
120 DATA 1,1,2,2,3,3,4,4,5,5
130 DATA4,4,5,5,6,6,7,7,8,8
```

```
ITEM #          STORE1          2               3               4
 1               1              10               1               4
 2               2               9               1               4
 3               3               8               2               5
 4               4               7               2               5
 5               5               6               3               6
 6               6               5               3               6
 7               7               4               4               7
 8               8               4               4               7
 9               9               2               5               8
 10             10               1               5               8
TOTALS          55              56              30              60
```

40 column display version.

```
10  REM *** SHOPPING GUIDE
20  DIM P(10,4),T(4)
30  REM *** READ DATA INTO ARRAYS
40  FOR S = 1 TO 4
50  FOR I = 1 TO 10: READ P(I,S): NEXT I: NEXT S
60  REM ***DISPLAY ROUTINE
70  PRINT "ITEM#"; TAB( 8);"STORE#1"; TAB( 16);"#2"; TAB( 24);"#3";
    TAB( 32);"#4": PRINT
80  FOR I = 1 TO 10: PRINT I;: LET M = 8
90  FOR S = 1 TO 4: PRINT  TAB( M);P(I,S);: LET T(S) = T(S) + P(I,S):
    LET M = M + 8: NEXT S
100  PRINT
110  NEXT I
120  LET M = 8: PRINT "TOTAL";: FOR X = 1 TO 4: PRINT  TAB( M);T(X);:
     LET M = M + 8: NEXT X
130  DATA 1,2,3,4,5,6,7,8,9,10
140  DATA 10,9,8,7,6,5,4,3,2,1
150  DATA 1,1,2,2,3,3,4,4,5,5
160  DATA 4,4,5,5,6,6,7,7,8,8
```

MORE HELPFUL WORDS FOR YOU

Computers for Everybody

Jerry Willis and Merl Miller

This fun-to-read book covers all the things you should know about computers. If you're anxious to buy one, use one or just want to find out about them, read this book first.

ISBN 0-918398-49-5

$5.95

Peanut Butter and Jelly Guide to Computers

Jerry Willis

This entertaining book is a simple, easy-to-digest source of information on personal computing. It leads you through all the essential knowledge of "computer literacy."

ISBN 0-918398-13-4

$9.95

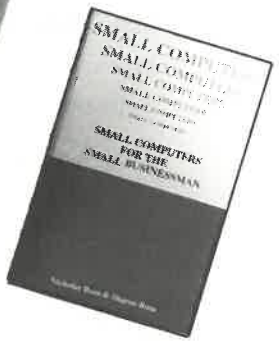
Nailing Jelly to a Tree

Jerry Willis and William Danley, Jr.

This is a book about software. The emphasis is on learning to use the thousands of available programs that have already been written, and adapting them to your machine.

ISBN 0-918398-42-8

$15.95



Small Computers for the Small Businessman

Nicholas Rosa and Sharon Rosa

If you've ever considered a computer for your business but didn't know where to turn, this is the book that will arm you with all the information you'll need to make an intelligent, cost-effective decision.

ISBN 0-918398-31-2

$16.95

dilithium Press, P.O. Box 606, Beaverton, OR 97075

Send to: dilithium Press, P.O. Box 606, Beaverton, OR 97075

Please send me the book(s) I have checked. I understand that if I'm not fully satisfied, I can return the book(s) within 10 days for full and prompt refund.

____ Computers for Everybody

____ Nailing Jelly to a Tree

____ Peanut Butter and Jelly Guide to Computers

____ Small Computers for the Small Businessman

☐ Check enclosed $_____
Payable to dilithium Press

☐ Send me your catalog of books.

☐ Please charge my
☐ VISA ☐ Mastercharge

_____ Exp. Date _____

Signature _____

Name _____

Address _____

City, State, Zip _____

1B